

QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 12, 2017

M. Bishop
Microsoft
February 8, 2017

Header Compression for HTTP/QUIC
draft-bishop-quic-http-and-qpac-02

Abstract

HTTP/2 [RFC7540] uses HPACK [RFC7541] for header compression. However, HPACK relies on the in-order message-based semantics of the HTTP/2 framing layer in order to function. Messages can only be successfully decoded if processed by the decoder in the same order as generated by the encoder. This draft refines HPACK to loosen the ordering requirements for use over QUIC [I-D.ietf-quic-transport].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 12, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	3
2. QPACK	3
2.1. Basic model	3
2.2. Changes to Static and Dynamic Tables	4
2.2.1. Changes to Header Table Size	4
2.2.2. Dynamic Table State Synchronization	5
2.3. Format of Header Management stream	6
2.3.1. Insert	6
2.3.2. Delete	7
2.3.3. Delete-Ack	10
2.4. Format of Encoded Headers on Message Streams	10
2.4.1. Indexed Header Field Representation	10
2.4.2. Literal Header Field Representation	11
3. Use in HTTP/QUIC	12
4. Performance Considerations	12
5. Security Considerations	13
6. IANA Considerations	13
7. Acknowledgements	13
8. Normative References	14
Author's Address	14

1. Introduction

HPACK has a number of features that were intended to provide performance advantages to HTTP/2, but which don't live well in an out-of-order environment such as that provided by QUIC.

The largest challenge is the fact that elements are referenced by a very fluid index. Not only is the index implicit when an item is added to the header table, the index will change without notice as other items are added to the header table. Static entries occupy the first 61 values, followed by dynamic entries. A newly-added dynamic entry would cause older dynamic entries to be evicted, and the retained items are then renumbered beginning with 62. This means that, without processing all preceding header sets, no index into the dynamic table can be interpreted, and the index of a given entry cannot be predicted.

Any solution to the above will almost certainly fall afoul of the memory constraints the decompressor imposes. The automatic eviction of entries is done based on the compressor's declared dynamic table

size, which **MUST** be less than the maximum permitted by the decompressor (and relayed using an HTTP/2 SETTINGS value).

Further, streams in QUIC are lossy in the presence of stream resets. While HTTP/2 (via TCP) guarantees the delivery of all previously-sent data on a stream even if that stream is reset, QUIC does not retransmit lost frames if a stream has been reset, and may discard data which has not yet been delivered to the application.

Previous versions of QPACK were small deltas of HPACK to introduce order-resiliency. This version departs from HPACK more substantially to add resilience against reset message streams.

In the following sections, this document proposes a new version of HPACK which makes different trade-offs, enabling partial out-of-order interpretation and bounded memory consumption with minimal head-of-line blocking. None of the proposed improvements to HPACK (strongly-typed fields, binary compression of common header syntax) are currently included, but certainly could be.

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119] and indicate requirement levels for compliant implementations.

2. QPACK

2.1. Basic model

HPACK combines header table modification and message header emission in a single sequence of coded bytes. QPACK bifurcates these into two channels:

- o A connection-wide series of table update instructions sent on a dedicated headers stream
- o Non-modifying instructions which use the current header table state to encode message headers

Because the per-message instructions introduce no changes to the header table state, no state is lost if these instructions are discarded due to a stream reset.

2.2. Changes to Static and Dynamic Tables

QPACK uses two tables for associating header fields to indexes. The static table is unchanged from [RFC7541].

The dynamic table is a map from index to header field. Indices are arbitrary numbers greater than the last index of the static table and less than 2^{27} . Each insert instruction will specify the index being modified. While any index MAY be chosen for a new entry, smaller numbers will yield better compression performance.

The dynamic table is still constrained to the size specified by the decoder. An attempt to add a header to the dynamic table which causes it to exceed the maximum size MUST be treated as an error by a decoder. To enable encoders to reclaim space, encoders can delete entries in the dynamic table, but can only reuse the index or the space after receiving confirmation of a successful deletion.

Because it is possible for QPACK frames to arrive which reference indices which have not yet been defined, such frames MUST wait until another frame has arrived and defined the index. In order to guard against malicious peers, implementations SHOULD impose a time limit and treat expiration of the timer as a decoding error. However, if the implementation chooses not to abort the connection, the remainder of the header block MUST be decoded and the output discarded.

2.2.1. Changes to Header Table Size

HTTP/QUIC prohibits mid-stream changes of settings. As a result, only one table size change is possible: From the value a client assumes during the 0-RTT flight to the actual value included in the server's SETTINGS frame. The assumed value is required to be either a server's previous value or zero. A server whose configuration has recently changed MAY overlook inadvertent violations of its maximum table size during the first round-trip.

In the case that the value has increased, either from zero to a non-zero value or from the cached value to a higher value, no action is required by the client. The encoder can simply begin using the additional space. In the case that the value has decreased, the encoder MUST immediately emit delete instructions which, upon completion, would bring the table within the required size.

Regardless of changes to header table size, the encoder MUST NOT add entries to the table which would result in a size greater than the maximum permitted. This can imply that no additions are permitted while waiting for these delete instructions to complete.

2.2.2. Dynamic Table State Synchronization

In order to ensure table consistency, all modifications of the header table occur on a dedicated control stream. Message control streams contain only indexed and literal header entries.

No entries are automatically evicted from the dynamic table. Size management is purely the responsibility of the encoder, which **MUST** NOT exceed the declared memory size of the decoder.

The encoder **SHOULD** track the following information about each entry in the table:

- o The list of recently-active streams which reference the entry in a trailer block, if any
- o The list of recently-active streams which reference the entry in a non-trailer block, if any

"Recently-active" streams are those which are still open or were closed less than a reasonable number of RTTs ago. An implementation **MAY** vary its definition of "recent" to trade off memory consumption and timely completion of deletes.

The encoder **MUST** consider memory as committed beginning when the indexed entry is assigned.

When the encoder wishes to delete an inserted value, it flows through the following set of states:

1. ***Delete requested.*** The encoder emits a delete instruction indicating which streams might have referenced the entry. The encoder **MUST NOT** reference the entry in any subsequent frame until this state machine has completed and **MUST** continue to include the entry in its calculation of consumed memory.
2. ***Delete pending.*** The decoder receives the delete instruction and checks the current state of its incoming streams (see Section 2.3.2.2). If more references might arrive, it stores the streams still needed and waits for them to complete.
3. ***Delete acknowledged.*** The decoder has received all QPACK frames which reference the deleted value, and can safely delete the entry. The decoder **SHOULD** promptly emit a Delete-Ack instruction on the header management stream.
4. ***Delete completed.*** When the encoder receives a Delete-Ack instruction acknowledging the delete, it no longer counts the

size of the deleted entry against the table size and MAY emit insert instructions for the field with a new value.

2.3. Format of Header Management stream

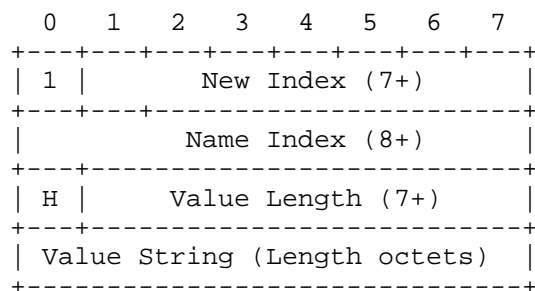
The header management stream contains a series of QPACK instructions with no message boundaries. Data on this stream SHOULD be processed as soon as it arrives.

This section describes the instructions which are possible on the Header Management stream.

2.3.1. Insert

An addition to the header table starts with the '1' one-bit pattern, followed by the new index of the header represented as an integer with a 7-bit prefix. This value is always greater than the number of entries in the static table.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with an 8-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Insert Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 8-bit index, followed by the header field name.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
	1		New Index (7+)				
+	+	+	+	+	+	+	+
	0						
+	+	+	+	+	+	+	+
	H		Name Length (7+)				
+	+	+	+	+	+	+	+
	Name String (Length octets)						
+	+	+	+	+	+	+	+
	H		Value Length (7+)				
+	+	+	+	+	+	+	+
	Value String (Length octets)						
+	+	+	+	+	+	+	+

Insert Header Field -- New Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

An encoder **MUST NOT** attempt to place a value at an index not known to be vacant. A decoder **MUST** treat the attempt to insert into an occupied slot as a fatal error.

2.3.2. Delete

A deletion from the header table starts with the '00' two bit pattern, followed by the index of the affected entry represented as an integer with a 6-bit prefix. This value is always greater than the number of entries in the static table.

A delete instruction then encodes a series of stream IDs which might have contained references to the entry in question.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
	0		0		Index (6+)		
+	+	+	+	+	+	+	+
	Non-Trailer List (*)						...
+	+	+	+	+	+	+	+
	Trailer List (*)						...
+	+	+	+	+	+	+	+

Delete Instruction

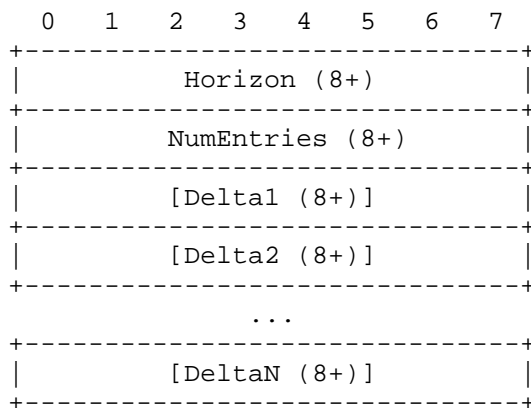
Both the Non-Trailer List and Trailer List are Stream ID Lists (see below) encoding a list of streams which might have referenced the entry either in non-trailer or trailer blocks.

2.3.2.1. Stream ID List

A Stream ID List encodes a sequence of stream IDs in two parts: First, a Horizon value indicates the first non-occurrence about which data is maintained. If data is maintained from the beginning of the connection, the Horizon is zero. This allows senders to succinctly express both old state which has been discarded and large regions where many or all streams contain references.

Following the horizon, a sequence of deltas indicates all streams since the Horizon on which a value has been used.

In the simplest case, a Stream ID List might be a horizon value followed by one zero byte. This indicates an absolute cut-off after which the entry is guaranteed not to be referenced.



Stream ID List

The field are as follows:

Horizon: The ID of the first stream for which the sender retains state which does not reference the deleted entry in the indicated block

NumEntries: The number of streams greater than the Horizon which might reference the entry and are listed in the remainder of the instruction

Delta1..N: A sequence of streams greater than the Horizon which might reference the entry, encoded as the difference in stream number from the previously-listed stream. This field is repeated NumEntries times.

2.3.2.2. Delete Validation

In order to safely delete an entry, a decoder MUST ensure that all outstanding references have arrived and been processed. Because no data is available about stream IDs less than the Horizon, a decoder MUST assume that any earlier stream ID might have contained a reference to the value in question.

A decoder can ensure all outstanding references have been processed by verifying that the following statements are true:

- o In the Non-Trailer Block, all streams less than the Horizon and all streams explicitly listed are in one of two states:
 - * closed
 - * headers completely processed
- o In the Trailer Block, all streams less than the Horizon and all streams explicitly listed are in one of three states:
 - * closed
 - * headers completely processed AND no trailers are expected
 - * trailers completely processed

An implementation MAY omit the "trailers completely processed" case, since the stream is expected to close immediately after receipt of the trailers block.

If these conditions are not met upon receipt of a Delete instruction, a decoder MUST wait to emit a Delete-Ack instruction until the outstanding streams have reached an appropriate state.

Note that a decoder MAY condense the list of specified streams by increasing the Horizon value and discarding those explicitly-listed stream IDs which are less than the new Horizon it has chosen. This delays delete completion, but reduces the amount of state to be tracked by the decoder without changing the correctness of the requirements above.

2.3.3. Delete-Ack

Confirmation that a delete has completed is expressed by an instruction which starts with the '01' two-bit pattern, followed by the index of the affected entry represented as an integer with a 6-bit prefix. This value is always greater than the number of entries in the static table.

Note that unlike all other instructions, this instruction refers to the receiver's dynamic table, not the sender's.

0	1	2	3	4	5	6	7
+---+---+---+---+---+---+---+---+							
0	1	Index (6+)					
+---+---+---+---+---+---+---+---+							

Delete-Ack Instruction

This instruction MUST NOT be sent before the conditions described in Section 2.3.2.2 have been satisfied, and SHOULD be sent as soon as possible once they are.

2.4. Format of Encoded Headers on Message Streams

Frames which carry HTTP message headers encode them using the following instructions:

2.4.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table and causes that header field to be added to the decoded header list, as described in Section 3.2 of [RFC7541].

0	1	2	3	4	5	6	7
+---+---+---+---+---+---+---+---+							
1	Index (7+)						
+---+---+---+---+---+---+---+---+							

Indexed Header Field

An indexed header field starts with the '1' 1-bit pattern, followed by the index of the matching header field, represented as an integer with a 7-bit prefix (see Section 5.1 of [RFC7541]).

The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

2.4.2. Literal Header Field Representation

A literal header field representation starts with the '0' 1-bit pattern and causes a header field to be added the decoded header list.

The second bit, 'N', indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the 'N' bit is set, the encoded header **MUST** always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the 'N' bit set, it **MUST** use the same representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 of [RFC7541] for more details).

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the index of the entry is represented as an integer with a 6-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
0	N	Name Index (6+)					
+	+	+	+	+	+	+	+
H	Value Length (7+)						
+	+	+	+	+	+	+	+
	Value String (Length octets)						
+	+	+	+	+	+	+	+

Literal Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 6-bit index, followed by the header field name.

0	1	2	3	4	5	6	7
0	N			0			
H				Name Length (7+)			
				Name String (Length octets)			
H				Value Length (7+)			
				Value String (Length octets)			

Literal Header Field -- Literal Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2).

3. Use in HTTP/QUIC

HTTP/QUIC [I-D.ietf-quic-http] currently retains the HPACK encoder/decoder from HTTP/2, using a Sequence number to enforce ordering. Using QPACK instead would entail the following changes:

- o The Sequence field is removed from HEADERS frames (Section 5.2.2) and PUSH_PROMISE frames (Section 5.2.6).
- o Header Block Fragments consist of QPACK data instead of HPACK data.
- o An additional control stream is reserved for header table updates. Alternately, this could be carried by HEADERS frames on the connection control stream.

A HEADERS or PUSH_PROMISE frame MAY contain an arbitrary number of QPACK instructions, but QPACK instructions SHOULD NOT cross a boundary between successive HEADERS frames. A partial HEADERS or PUSH_PROMISE frame MAY be processed upon arrival and the resulting partial header set emitted or buffered according to implementation requirements.

4. Performance Considerations

While QPACK is designed to minimize head-of-line blocking between streams on header decoding, there are some situations in which lost or delayed packets can still impact the performance of header compression.

References to indexed entries will block if the frame containing the entry definition is lost or delayed. Encoders MAY choose to trade off compression efficiency and avoid blocking by using literal instructions rather than referencing the dynamic table until the insertion is believed to be complete.

Since it is possible to insert header values without emitting them on a stream, an encoder MAY proactively insert header values which it believes will be needed on future requests.

Delayed frames which prevent deletes from completing can prevent the encoder from adding any new entries due to the maximum table size. This does not block the encoder from continuing to make requests, but could sharply limit compression performance. Encoders would be well-served to delete entries in advance of encountering the table maximum. Decoders SHOULD be prompt about emitting Delete-Ack instructions to enable the encoder to recover the table space.

5. Security Considerations

A malicious encoder might attempt to consume a large amount of space on the decoder by opening the maximum number of streams, adding entries to the table, then sending delete instructions enumerating many streams in a Stream ID List.

To guard against such attacks, a decoder SHOULD bound its state tracking by generalizing the list of streams to be tracked. This is most easily achieved by advancing the Horizon to a later value and discarding explicit Stream IDs to track, but can also be accomplished by eliding explicit streams in ranges. This does not cause any loss of consistency for deletes, but could delay completion and reduce performance if done aggressively.

6. IANA Considerations

This document currently makes no request of IANA, and might not need to.

7. Acknowledgements

This draft draws heavily on the text of [RFC7541]. The indirect input of those authors is gratefully acknowledged, as well as ideas gleefully stolen from:

- o Jana Iyengar
- o Patrick McManus

- o Martin Thomson
- o Charles 'Buck' Krasic
- o Kyle Rose

8. Normative References

- [I-D.ietf-quick-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quick-http-01 (work in progress), January 2017.
- [I-D.ietf-quick-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quick-transport-01 (work in progress), January 2017.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<http://www.rfc-editor.org/info/rfc7540>>.
- [RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<http://www.rfc-editor.org/info/rfc7541>>.

Author's Address

Mike Bishop
Microsoft

Email: michael.bishop@microsoft.com

QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 17, 2018

M. Bishop
Akamai
December 14, 2017

Header Compression for HTTP/QUIC
draft-bishop-quic-http-and-qpak-07

Abstract

HTTP/2 [RFC7540] uses HPACK [RFC7541] for header compression. However, HPACK relies on the in-order message-based semantics of the HTTP/2 framing layer in order to function. Messages can only be successfully decoded if processed by the decoder in the same order as generated by the encoder. This draft refines HPACK to loosen the ordering requirements for use over QUIC [I-D.ietf-quic-transport].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 17, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	3
2. QPACK - Concepts	3
2.1. Changes to Static and Dynamic Tables	4
2.1.1. Dynamic Table State Synchronization	4
2.2. Encoding Constraints	6
2.2.1. Permitted References	6
2.2.2. Header Table Size	6
3. Wire Format	7
3.1. Feedback Stream	8
3.1.1. HEADERS_DONE	8
3.1.2. ACK_FLUSH	8
3.1.3. DROP	9
3.1.4. ACK_DROP	9
3.2. Checkpoint Streams	10
3.2.1. INSERT	10
3.2.2. TOUCH	12
3.3. Request Streams	12
3.3.1. Indexed Header Field Representation	13
3.3.2. Literal Header Field Representation	13
4. Use in HTTP/QUIC	14
4.1. SETTING_QPACK_BLOCKING_PERMITTED	15
4.2. SETTING_QPACK_INITIAL_CHECKPOINT	15
5. Implementation trade-offs	15
5.1. Compression Efficiency versus Blocking Avoidance	16
5.2. Timely State Transitions versus Decoder Complexity	16
6. Security Considerations	17
7. IANA Considerations	17
7.1. Settings	17
7.2. Errors	18
8. Acknowledgements	18
9. References	18
9.1. Normative References	18
9.2. Informative References	19
Author's Address	19

1. Introduction

HPACK has a number of features that were intended to provide performance advantages to HTTP/2, but which don't live well in an out-of-order environment such as that provided by QUIC.

The largest challenge is the fact that elements are referenced by a very fluid index. Not only is the index implicit when an item is added to the header table, the index will change without notice as other items are added to the header table. Static entries occupy the first 61 values, followed by dynamic entries. A newly-added dynamic entry would cause older dynamic entries to be evicted, and the retained items are then renumbered beginning with 62. This means that, without processing all preceding header sets, no index into the dynamic table can be interpreted, and the index of a given entry cannot be predicted.

Any solution to the above will almost certainly fall afoul of the memory constraints the decompressor imposes. The automatic eviction of entries is done based on the compressor's declared dynamic table size, which **MUST** be less than the maximum permitted by the decompressor (and relayed using an HTTP/2 SETTINGS value).

Further, streams in QUIC are lossy in the presence of stream resets. While HTTP/2 (via TCP) guarantees the delivery of all previously-sent data on a stream even if that stream is reset, QUIC does not retransmit lost frames if a stream has been reset, and may discard data which has not yet been delivered to the application.

Early versions of QPACK were small deltas of HPACK to introduce order-resiliency. Recent versions depart from HPACK more substantially to add resilience against reset message streams and reduce the impact of head-of-line blocking.

In the following sections, this document proposes a successor to HPACK which makes different trade-offs, enabling partial out-of-order interpretation and bounded memory consumption with minimal head-of-line blocking. None of the proposed improvements to HPACK (strongly-typed fields, binary compression of common header syntax) are currently included, but certainly could be.

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119] and indicate requirement levels for compliant implementations.

2. QPACK - Concepts

HPACK combines header table modification and message header emission in a single sequence of coded bytes. QPACK bifurcates these into three channels:

- o Connection-wide sets of table update instructions sent on non-request streams
- o Connection-wide feedback on stream and checkpoint state on a single non-request stream
- o Non-modifying instructions which use the current header table state to encode message headers on request streams

Because the per-message instructions introduce no changes to the header table state, no state is lost if these instructions are discarded due to a stream reset. Because the updates to the header table supply their own order controls (the checkpoint logic), they can be processed in any order and therefore delivered as messages using unidirectional QUIC streams.

2.1. Changes to Static and Dynamic Tables

QPACK uses two tables for associating header fields to indexes. The static table is unchanged from [RFC7541]. Unlike in [RFC7541], the tables are not concatenated, but are referenced separately.

The dynamic table is a map from index to header field. Indices are arbitrary numbers between 1 and 2^{27} . Each insert instruction will specify the index being modified. While any index MAY be chosen for a new entry, smaller numbers will yield better compression performance.

With decoder consent (see Section 4.1), it is possible for QPACK instructions to arrive which reference indices which have not yet been defined. Such instructions MUST wait until the index definition has arrived. In order to guard against malicious peers, implementations supporting blocking SHOULD impose a time limit and treat expiration of the timer as a decoding error.

2.1.1. Dynamic Table State Synchronization

In order to ensure table consistency, all modifications of the header table occur as separate messages rather than on request streams. Request streams contain only indexed and literal header entries.

No entries are automatically evicted from the dynamic table. Size management is purely the responsibility of the encoder, which MUST NOT exceed the declared memory size of the decoder.

To simplify state management in the dynamic table, `_checkpoints_` are introduced. A checkpoint is used to track entries added to the dynamic table and streams that reference those entries, rather than

maintaining the full state of which streams reference which table entries.

Checkpoints are unordered and have an identifier which **MUST** be unique among checkpoints which have not been dropped. Each checkpoint has a unidirectional stream which begins with its identifier and contains a series of updates associated with that checkpoint. These updates **SHOULD** be processed as they arrive; it is not necessary (and might not be desirable) to wait for all instructions associated with a checkpoint to arrive before beginning to process it.

The feedback stream is used to relay state transitions to the peer. For example, when a decoder is done processing a header block, it signals this using the `HEADERS_DONE` message. The encoder uses this information to track which checkpoints can be dropped.

2.1.1.1. Checkpoint Lifecycle

A checkpoint is created by opening a new checkpoint stream. This places the checkpoint in the **NEW** state for both encoder and decoder. The encoder typically has at least one checkpoint in the **NEW** state.

Flushing a checkpoint is a two-step operation. First, the checkpoint stream is closed. At that time, the encoder's **NEW** checkpoint becomes **PENDING**. The decoder moves its **NEW** checkpoint directly to **LIVE** and responds with an `ACK_FLUSH` message on the feedback stream. When the encoder receives this message, its **PENDING** checkpoint becomes **LIVE**.

Unused entries are evicted indirectly, by dropping checkpoints. Before a checkpoint can be dropped, its state is changed to **DYING**. Changing a checkpoint's state to **DYING** allows the checkpoint to age out. This is a strictly internal state on the encoder, and not visible to the decoder. A **DYING** checkpoint can be returned to **LIVE** at the encoder's discretion if necessary.

The encoder can change a **DYING** checkpoint to **DEAD** (sending a `DROP` instruction) when it is no longer referenced by any outstanding header blocks. The encoder sends the `DROP` command to the decoder when it declares a checkpoint **DEAD**.

To ensure consistency, the decoder drops the corresponding checkpoint and responds with an `ACK_DROP` message only when it has fully received all instructions the encoder has issued up to that point. The encoder drops the **DEAD** checkpoint upon receipt of the `ACK_DROP` message.

When a checkpoint is dropped by encoder or decoder, the table entries it references are checked: if an entry is no longer referenced by any checkpoint, the entry is evicted.

Dropping a checkpoint and the entries associated with it is not limited to just the oldest checkpoint; any DYING checkpoint - as long as state transition rules are followed - may be dropped. This flexibility permits the encoder to use a number of strategies for entry eviction.

As long as the maximum dynamic table size is observed, new checkpoints can be created; no upper limit on the number of checkpoints is specified. A well-balanced spread of checkpoints permits the encoder to recycle entries effectively.

2.2. Encoding Constraints

2.2.1. Permitted References

When encoding headers on a request stream, an encoder MAY reference any static table entry or any dynamic header table entry referenced by a LIVE checkpoint. References to entries in NEW or PENDING checkpoints are permitted only if the client has set "SETTING_QPACK_BLOCKING_PERMITTED" (see Section 4.1).

If a decoder receives a reference to an empty slot in the dynamic table but has not sent "SETTING_QPACK_BLOCKING_PERMITTED", this MUST be treated as a stream error of type "ERROR_QPACK_INVALID_REFERENCE" if on a request stream. References to empty slots in the dynamic table on a checkpoint stream MUST be treated as a connection error of type "ERROR_QPACK_INVALID_REFERENCE".

References to DYING checkpoints are possible by returning the checkpoint to LIVE, but this is usually inadvisable. Table entries contained only in a DEAD checkpoint can never be referenced.

2.2.2. Header Table Size

As in HPACK, the dynamic table is constrained to the maximum size specified by the decoder. An attempt to add a header to the dynamic table or to create a new checkpoint which causes it to exceed the maximum size MUST be treated as an error by a decoder. To enable encoders to reclaim space, encoders can drop old checkpoints (see Section 2.1.1).

The total table size is calculated as follows:

- o The size of each entry is calculated as in HPACK

- o Each checkpoint that has not been removed, regardless of state, consumes 64 bytes

2.2.2.1. Table Size Changes

HTTP/QUIC prohibits mid-stream changes of settings. As a result, only one table size change is possible: From the value a client assumes during the 0-RTT flight to the actual value included in the server's SETTINGS frame. The assumed value is required to be either a server's previous value or zero. A server whose configuration has recently changed MAY overlook inadvertent violations of its maximum table size during the first round-trip.

In the case that the value has increased, either from zero to a non-zero value or from the cached value to a higher value, no action is required by the client. The encoder can simply begin using the additional space. In the case that the value has decreased, the encoder MUST move checkpoints to the DYING state which, upon removal, would bring the table within the required size.

Regardless of changes to header table size, the encoder MUST NOT create new checkpoints or add entries to the table which would result in a size greater than the maximum permitted. This can imply that no additions are permitted while waiting for old checkpoints to complete.

3. Wire Format

QPACK instructions occur on three stream types, each of which uses a separate instruction space.

The feedback stream is a bidirectional server-initiated stream used for acknowledgement of actions and checkpoint state management. Checkpoint streams are unidirectional streams from encoder to decoder. Both types of streams consist of a series of QPACK instructions with no message boundaries, preceded by a stream header for checkpoint streams.

Finally, the contents of HEADERS and PUSH_PROMISE frames on request streams reference the QPACK table state.

This section describes the instructions which are possible on each stream type.

3.1. Feedback Stream

Stream 1, the first server-initiated bidirectional stream, is used as the feedback stream, since the client does not need to begin sending data on this stream until it has received data from the server.

This stream is critical to the HTTP/QUIC connection, and carries a stream of the instructions defined in this section. Data on this stream **SHOULD** be processed as soon as it arrives.

3.1.1. HEADERS_DONE

When the decoder has processed a frame containing header emission instructions (Section 3.3, HEADERS or PUSH_PROMISE frames) on a stream, it **MUST** emit a HEADERS_DONE message on the feedback stream. The same Stream ID can be identified multiple times, as multiple header-containing blocks can be sent on a single stream in the case of intermediate responses, trailers, pushed requests, etc.

Since header frames on a request stream are received and processed in order, this gives the encoder precise feedback on which header blocks within a stream have been fully processed. This information can then be used to correctly track outstanding stream references to checkpoints.

```

    0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 1 |           Stream ID (7+) |
+---+---+---+---+---+---+---+

```

HEADERS_DONE instruction

3.1.2. ACK_FLUSH

When the decoder has finished processing all instructions that make up a checkpoint, it **MUST** indicate successful processing to the encoder by emitting an ACK_FLUSH instruction on the feedback stream.

Upon emitting an ACK_FLUSH, the checkpoint transitions from NEW to LIVE on the decoder. Upon receipt of an ACK_FLUSH, the checkpoint transitions from PENDING to LIVE on the encoder.

```

    0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 1 | 0 | Checkpoint ID (5+) |
+---+---+---+---+---+---+---+

```

ACK_FLUSH instruction

3.1.3. DROP

When an encoder has received sufficient HEADERS_DONE messages to know that a DYING checkpoint has no outstanding references, it emits a DROP instruction to inform the decoder that the checkpoint can be removed. Upon sending a DROP instruction, a DYING checkpoint becomes DEAD. The DROP instruction also includes the IDs of any PENDING or NEW checkpoints which reference entries contained in the checkpoint being dropped. The "L" bit in each byte indicates whether another checkpoint ID follows (L=0) or this is the final byte of the DROP instruction (L=1).

Upon receiving a DROP instruction, if all listed checkpoints have been fully processed (transitioned from NEW to LIVE), the identified LIVE checkpoint is immediately removed from the decoder state and an ACK_DROP instruction is emitted. Otherwise, the decoder saves the DROP instruction until other checkpoints become LIVE.

0	1	2	3	4	5	6	7
0	0	L	Checkpoint ID (5+)				
L	Checkpoint (7+)						
L	Checkpoint (7+)						
			...				

DROP instruction

3.1.4. ACK_DROP

When a decoder receives a DROP instruction, it removes the referenced checkpoint from its state and clears any table entries which were referenced only by that checkpoint. It then emits an ACK_DROP instruction. When an encoder receives an ACK_DROP instruction, it removes the corresponding DEAD checkpoint from its state and clears any table entries which were referenced only by that checkpoint.

0	1	2	3	4	5	6	7
0	1	1	Checkpoint ID (5+)				

ACK_DROP instruction

3.2. Checkpoint Streams

Each checkpoint stream indicates the creation and content of a NEW checkpoint. Each checkpoint has an ID; these IDs are chosen arbitrarily by the encoder, though lower values SHOULD be preferred. IDs of checkpoints which have been dropped MAY be reused for future NEW checkpoints.

When the encoder has finished writing all data on the stream, it changes the checkpoint to PENDING. When the decoder has received and processed all data on the stream, it changes the checkpoint to LIVE and generates an ACK_FLUSH.

Unidirectional streams in HTTP/QUIC begin with a stream header indicating the nature of the stream content; the identifier for QPACK checkpoints is 0x4B.

Note to readers: This header does not currently exist in the main draft, but has manifested in several PRs, and would need to be resurrected.

Following the stream header, a checkpoint stream contains its checkpoint ID as an 8-bit prefix integer. The remainder of the stream's data consists of the instructions defined in this section.

Data on checkpoint streams SHOULD be processed as soon as it arrives. If multiple checkpoint streams are received at once, a decoder SHOULD process data on each as it arrives if it has sent "SETTINGS_QPACK_BLOCKING_PERMITTED", but MAY process checkpoint streams one at a time.

3.2.1. INSERT

An addition to the dynamic table starts with the '1' one-bit pattern, followed by the new index of the header represented as an integer with a 7-bit prefix. The decoder adds the supplied header to the checkpoint currently being processed, which is in the NEW state.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the "S" bit indicates whether the reference is to the static (S=1) or dynamic (S=0) table and the index of the entry is represented as an integer with an 7-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.

If an INSERT instruction uses an existing dynamic table entry for the name of an entry being added to the NEW checkpoint, both the existing

entry and the new entry are referenced by the NEW checkpoint. INSERT instructions which reference the dynamic table MUST reference only entries which are already included in a LIVE checkpoint. This avoids the possibility of one checkpoint stream blocking on a different checkpoint.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
	1		New Index (7+)				
+	+	+	+	+	+	+	+
	S		Name Index (7+)				
+	+	+	+	+	+	+	+
	H		Value Length (7+)				
+	+	+	+	+	+	+	+
	Value String (Length octets)						
+	+	+	+	+	+	+	+

INSERT instruction -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the table reference, followed by the header field name.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
	1		New Index (7+)				
+	+	+	+	+	+	+	+
	0						
+	+	+	+	+	+	+	+
	H		Name Length (7+)				
+	+	+	+	+	+	+	+
	Name String (Length octets)						
+	+	+	+	+	+	+	+
	H		Value Length (7+)				
+	+	+	+	+	+	+	+
	Value String (Length octets)						
+	+	+	+	+	+	+	+

INSERT instruction -- New Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

An encoder MUST NOT attempt to place a value at an index not known to be vacant. A decoder MUST treat the attempt to insert into an occupied slot or reference a name in a vacant slot as a fatal error.

3.2.2. TOUCH

This instruction is emitted to link a NEW checkpoint to an existing header table entry created by a previous checkpoint. This causes the entry not to be removed from the table so long as the current checkpoint is alive.

```

      0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 |           Index (7+)          |
+---+-----+

```

Indexed Header Field

The encoder SHOULD NOT issue multiple TOUCH commands for the same entry in the context of the same NEW checkpoint. If a non-existent index is specified, the decoder MUST treat it as an error.

3.3. Request Streams

Frames which carry HTTP message headers begin with an optional preface indicating potentially-blocking references in the frame. If present, this preface indicates that the request depends on one or more checkpoints which were NEW or PENDING for the encoder when the frame was generated. If these checkpoints are not LIVE on the decoder, it MAY delay reading the remainder of the frame until they are. (If any of these checkpoints have already been dropped, this must be treated as a stream error of type `ERROR_QPACK_INVALID_REFERENCE`.)

The preface is formatted as follows:

```

      0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| L |           Checkpoint (7+)          |
+---+---+---+---+---+---+---+---+
| L |           Checkpoint (7+)          |
+---+---+---+---+---+---+---+---+
|           ...                          |
+---+---+---+---+---+---+---+---+

```

QPACK preface

The "L" bit indicates that this checkpoint is the last checkpoint in the preface; if the bit is unset (0), then another checkpoint follows.

3.3.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table and causes that header field to be added to the decoded header list, as described in Section 3.2 of [RFC7541].

0	1	2	3	4	5	6	7
+---+---+---+---+---+---+---+---+							
1	S	Index (6+)					
+---+---+---+---+---+---+---+---+							

Indexed Header Field

An indexed header field starts with the '1' 1-bit pattern, followed by the "S" bit indicating whether the reference is into the static (S=1) or dynamic (S=0) table. Finally, the index of the matching header field is represented as an integer with a 6-bit prefix (see Section 5.1 of [RFC7541]).

The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

3.3.2. Literal Header Field Representation

A literal header field representation starts with the '0' 1-bit pattern and causes a header field to be added the decoded header list.

The second bit, 'N', indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the 'N' bit is set, the encoded header MUST always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the 'N' bit set, it MUST use the same representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 of [RFC7541] for more details).

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the "S" bit indicates whether the reference is to the static (S=1) or dynamic (S=0) table and the index of the entry is represented as an integer with an 5-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
0	N	S	Name Index (5+)				
+	+	+	+	+	+	+	+
H	Value Length (7+)						
+	+	+	+	+	+	+	+
	Value String (Length octets)						
+	+	+	+	+	+	+	+

Literal Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 6-bit index, followed by the header field name.

0	1	2	3	4	5	6	7
+	+	+	+	+	+	+	+
0	N	0					
+	+	+	+	+	+	+	+
H	Name Length (7+)						
+	+	+	+	+	+	+	+
	Name String (Length octets)						
+	+	+	+	+	+	+	+
H	Value Length (7+)						
+	+	+	+	+	+	+	+
	Value String (Length octets)						
+	+	+	+	+	+	+	+

Literal Header Field -- Literal Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

4. Use in HTTP/QUIC

HTTP/QUIC [I-D.ietf-quic-http] currently retains the HPACK encoder/decoder from HTTP/2, but restricts the size of the dynamic table to zero. Using QPACK instead would entail the following changes:

- o Header Blocks consist of QPACK data instead of HPACK data
- o HEADERS and PUSH_PROMISE frames define a flag indicating the presence of a preface.
- o Just as unidirectional push streams have a stream header identifying their Push ID, a header will need to be added to differentiate checkpoint streams from pushes

- o Stream 2 is reserved for the Feedback Stream

A HEADERS or PUSH_PROMISE frame MAY contain an arbitrary number of QPACK instructions. A partial HEADERS or PUSH_PROMISE frame MAY be processed upon arrival and the resulting partial header set emitted or buffered according to implementation requirements.

4.1. SETTING_QPACK_BLOCKING_PERMITTED

An HTTP/QUIC implementation can trade off the complexity of its QPACK decoder against compression efficiency by permitting the peer's compressor to reference unacknowledged entries. In the case of loss on a checkpoint stream, such references might cause the processing of request streams to block, waiting for the arrival of missing data.

If the decoder permits the encoder to make blocking references, it sets "SETTING_QPACK_BLOCKING_PERMITTED" (0xSETTING-TBD1) to a non-zero value. The encoder receiving this setting MAY encode up to this number of potentially-blocking references at a time.

Sending this setting with no value indicates that a decoder is willing to tolerate blocking references bounded only by the allowed number of streams. If a decoder does not send this setting or sends this setting with a value of zero, the encoder MUST NOT encode a header using a reference that might block.

4.2. SETTING_QPACK_INITIAL_CHECKPOINT

An HTTP/QUIC implementation MAY include the "SETTING_QPACK_INITIAL_CHECKPOINT" (0xSETTING-TBD2) setting, containing the full serialization of an initial checkpoint stream's data. If present, this setting MUST be fully processed by the peer before decoding any checkpoint streams or header frames on request streams.

The checkpoint defined by this setting is considered LIVE by both the encoder and the decoder from the beginning of the connection. The decoder does not need to send an ACK_FLUSH message confirming receipt of this setting.

5. Implementation trade-offs

This document specifies a means for the encoder to express the choices it made while encoding, but intentionally does not mandate what those choices should be. In this section, potential areas for implementation tuning are explored.

5.1. Compression Efficiency versus Blocking Avoidance

If blocking references are permitted, they will block if the frame containing the entry definition is lost or delayed. Encoders MAY choose to trade off compression efficiency and avoid blocking by using literal instructions rather than referencing the dynamic table until the insertion is believed to be complete.

The most efficient compression algorithm will reference a table entry whenever it exists in the table, but risks blocking when subject to packet loss or reordering. The most conservative algorithm will always emit literals to guarantee that no blocking will ever occur. Most implementations will choose a balance between these two extremes.

Better efficiency while being similarly conservative can be achieved by permitting references to table entries only once these entries are confirmed to be present in the table. More optimization can be achieved when the reference is known to be in the same packet as the definition.

Increases in efficiency can be achieved by assuming greater risk of blocking - implementations might choose a particular balance, or adjust their aggressiveness based on observed network characteristics.

Since it is possible to insert header values without emitting them on a stream, an encoder MAY also proactively insert header values which it believes will be needed on future requests, at the cost of reduced compression efficiency for incorrect predictions.

The ability to split updates to the header table into discrete checkpoints reduces the possibility for head-of-line blocking within the checkpoint streams. Implementations SHOULD limit the size of checkpoints to avoid head-of-line blocking within these messages.

5.2. Timely State Transitions versus Decoder Complexity

Anything which prevent checkpoints from transitioning from DYING to DEAD can prevent the encoder from adding any new entries due to the maximum table size. This does not block the encoder from continuing to make requests, but could sharply limit compression performance. Encoders would be well-served to begin moving checkpoint to DYING in advance of encountering the table maximum. Decoders SHOULD be prompt about emitting STREAM_DONE and ACK_DROP instructions to enable the encoder to recover the table space.

Similarly, for decoders which prohibit blocking references, delaying the transition of a checkpoint from PENDING to LIVE will degrade compression performance. Decoders SHOULD consume checkpoint data and emit ACK_FLUSH frames as promptly as possible.

Since decoders cannot safely drop old checkpoints until they have fully processed any checkpoints which might have been open concurrently, a long-lived checkpoint can delay the completion of an ACK_DROP. Encoders SHOULD flush all NEW checkpoints as soon as feasible after issuing a DROP instruction.

6. Security Considerations

A malicious encoder might attempt to consume a large amount of space on the decoder, but as each decoder chooses how much memory to allow the peer to consume, this state is bounded.

A malicious encoder might also send blocking references to entries which will never actually be defined. This attack is comparable to a "slow loris" attack in which a request is delivered very slowly in an attempt to consume resources on the server. Similar mitigations (request timers, etc.) SHOULD be employed to guard against such attacks.

7. IANA Considerations

This document registers two settings and one error code with the corresponding HTTP/QUIC registries.

7.1. Settings

This document registers two entries in the "HTTP/QUIC Settings" registry established by [I-D.ietf-quic-http].

Setting Name: SETTING_QPACK_BLOCKING_PERMITTED

Code: 0xSETTING-TBD1

Specification: Section 4.1

and

Setting Name: SETTING_QPACK_INITIAL_CHECKPOINT

Code: 0xSETTING-TBD2

Specification: Section 4.2

7.2. Errors

This document registers one error code in the "HTTP/QUIC Error Code" registry established by [I-D.ietf-quic-http].

Error name: ERROR_QPACK_INVALID_REFERENCE

Code: 0xERROR-TBD

Description: A blocking reference was received by a decoder which did not permit it

Specification: Section 2.2.1

8. Acknowledgements

This draft draws heavily on the text of [RFC7541], and adopts (with adaptation) the checkpoint model from [QMIN]. The direct and indirect input of those authors is gratefully acknowledged, as well as ideas gleefully stolen from:

- o Jana Iyengar
- o Patrick McManus
- o Martin Thomson
- o Charles 'Buck' Krasic
- o Kyle Rose
- o Alan Frindell

A substantial portion of Mike's work on this draft was supported by Microsoft during his employment there.

9. References

9.1. Normative References

- [I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-07 (work in progress), October 2017.

[I-D.ietf-quic-transport]

Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<https://www.rfc-editor.org/info/rfc7541>>.

9.2. Informative References

[QMIN] Tikhonov, D., "QMIN: Header Compression for QUIC", draft-tikhonov-quic-qmin-00 (work in progress), November 2017.

Author's Address

Mike Bishop
Akamai

Email: mbishop@evequefou.be

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Bishop, Ed.
Microsoft
March 13, 2017

Hypertext Transfer Protocol (HTTP) over QUIC
draft-ietf-quic-http-02

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at <https://github.com/quicwg> ; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/http> .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. QUIC Advertisement	3
2.1. QUIC Version Hints	4
3. Connection Establishment	4
3.1. Draft Version Identification	5
4. Stream Mapping and Usage	5
4.1. Stream 3: Connection Control Stream	6
4.2. HTTP Message Exchanges	6
4.2.1. Header Compression	7
4.2.2. The CONNECT Method	8
4.3. Stream Priorities	9
4.4. Server Push	9
5. HTTP Framing Layer	10
5.1. Frame Layout	10
5.2. Frame Definitions	10
5.2.1. HEADERS	10
5.2.2. PRIORITY	11
5.2.3. SETTINGS	12
5.2.4. PUSH_PROMISE	15
6. Error Handling	15
6.1. HTTP-Defined QUIC Error Codes	16
7. Considerations for Transitioning from HTTP/2	17
7.1. HTTP Frame Types	17
7.2. HTTP/2 SETTINGS Parameters	18
7.3. HTTP/2 Error Codes	19
8. Security Considerations	20
9. IANA Considerations	21
9.1. Registration of HTTP/QUIC Identification String	21
9.2. Registration of QUIC Version Hint Alt-Svc Parameter	21
9.3. Existing Frame Types	21

9.4. Settings Parameters	22
9.5. Error Codes	23
10. References	25
10.1. Normative References	25
10.2. Informative References	26
Appendix A. Contributors	26
Appendix B. Change Log	26
B.1. Since draft-ietf-quic-http-01:	26
B.2. Since draft-ietf-quic-http-00:	27
B.3. Since draft-shade-quic-http2-mapping-00:	27
Author's Address	27

1. Introduction

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC, drawing heavily on the existing TCP mapping, HTTP/2. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [RFC7540].

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

2. QUIC Advertisement

An HTTP origin advertises the availability of an equivalent HTTP/QUIC endpoint via the Alt-Svc HTTP response header or the HTTP/2 ALTSVC frame ([RFC7838]), using the ALPN token defined in Section 3.

For example, an origin could indicate in an HTTP/1.1 or HTTP/2 response that HTTP/QUIC was available on UDP port 443 at the same hostname by including the following header in any response:

```
Alt-Svc: hq=":443"
```

On receipt of an Alt-Svc header indicating HTTP/QUIC support, a client MAY attempt to establish a QUIC connection to the indicated host and port and, if successful, send HTTP requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

2.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/QUIC clients. QUIC versions are four-octet sequences with no additional constraints on format. Syntax:

quic = version-number

version-number = 1*8HEXDIG; hex-encoded QUIC version

Leading zeros SHOULD be omitted for brevity. When multiple versions are supported, the "quic" parameter MAY be repeated multiple times in a single Alt-Svc entry. For example, if a server supported both version 0x00000001 and the version rendered in ASCII as "Q034", it could specify the following header:

```
Alt-Svc: hq=":443";quic=1;quic=51303334
```

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Origins SHOULD list only versions which are supported by the alternative, but MAY omit supported versions for any reason.

3. Connection Establishment

HTTP/QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/QUIC support is indicated by selecting the ALPN token "hq" in the crypto handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 5.2.3) MUST be sent as the initial frame of the HTTP control stream (StreamID 3, see Section 4). The server MUST NOT send data on any other stream until the client's SETTINGS frame has been received.

3.1. Draft Version Identification

***RFC Editor's Note:** Please remove this section prior to publication of a final version of this document.

Only implementations of the final, published RFC can identify themselves as "hq". Until such an RFC exists, implementations **MUST NOT** identify themselves using this string.

Implementations of draft versions of the protocol **MUST** add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "hq-01".

Non-compatible experiments that are based on these draft versions **MUST** append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 which reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "hq-09-rickroll". Note that any label **MUST** conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

4. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. A QUIC receiver buffers and orders received STREAM frames, exposing the data contained within as a reliable byte stream to the application.

QUIC reserves Stream 1 for crypto operations (the handshake, crypto config updates). Stream 3 is reserved for sending and receiving HTTP control frames, and is analogous to HTTP/2's Stream 0. This connection control stream is considered critical to the HTTP connection. If the connection control stream is closed for any reason, this **MUST** be treated as a connection error of type `QUIC_CLOSED_CRITICAL_STREAM`.

When HTTP headers and data are sent over QUIC, the QUIC layer handles most of the stream management. An HTTP request/response consumes a pair of streams: This means that the client's first request occurs on QUIC streams 5 and 7, the second on stream 9 and 11, and so on. The server's first push consumes streams 2 and 4. This amounts to the second least-significant bit differentiating the two streams in a request.

The lower-numbered stream is called the message control stream and carries frames related to the request/response, including HEADERS. The higher-numbered stream is the data stream and carries the request/response body with no additional framing. Note that a request or response without a body will cause this stream to be half-closed in the corresponding direction without transferring data.

Because the message control stream contains HPACK data which manipulates connection-level state, the message control stream **MUST** NOT be closed with a stream-level error. If an implementation chooses to reject a request with a QUIC error code, it **MUST** trigger a QUIC RST_STREAM on the data stream only. An implementation **MAY** close (FIN) a message control stream without completing a full HTTP message if the data stream has been abruptly closed. Data on message control streams **MUST** be fully consumed, or the connection terminated.

All message control streams are considered critical to the HTTP connection. If a message control stream is terminated abruptly for any reason, this **MUST** be treated as a connection error of type HTTP_RST_CONTROL_STREAM. When a message control stream terminates cleanly, if the last frame on the stream was truncated, this **MUST** be treated as a connection error (see HTTP_MALFORMED_* in Section 6.1).

Pairs of streams must be utilized sequentially, with no gaps. The data stream is opened at the same time as the message control stream is opened and is closed after transferring the body. The data stream is closed immediately after sending the request headers if there is no body.

HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction. Requests and responses are considered complete when the corresponding QUIC streams are closed in the appropriate direction.

4.1. Stream 3: Connection Control Stream

Since most connection-level concerns will be managed by QUIC, the primary use of Stream 3 will be for the SETTINGS frame when the connection opens and for PRIORITY frames subsequently.

4.2. HTTP Message Exchanges

A client sends an HTTP request on a new pair of QUIC streams. A server sends an HTTP response on the same streams as the request.

An HTTP message (request or response) consists of:

1. one header block (see Section 5.2.1) on the control stream containing the message headers (see [RFC7230], Section 3.2),
2. the payload body (see [RFC7230], Section 3.3), sent on the data stream,
3. optionally, one header block on the control stream containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

In addition, prior to sending the message header block indicated above, a response may contain zero or more header blocks on the control stream containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2).

The data stream **MUST** be half-closed immediately after the transfer of the body. If the message does not contain a body, the corresponding data stream **MUST** still be half-closed without transferring any data. The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] **MUST NOT** be used.

Trailing header fields are carried in an additional header block on the message control stream. Such a header block is a sequence of HEADERS frames with End Header Block set on the last frame. Senders **MUST** send only one header block in the trailers section; receivers **MUST** decode any subsequent header blocks in order to maintain HPACK decoder state, but the resulting output **MUST** be discarded.

An HTTP request/response exchange fully consumes a pair of streams. After sending a request, a client closes the streams for sending; after sending a response, the server closes its streams for sending and the QUIC streams are fully closed.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server **MAY** request that the client abort transmission of a request without error by sending a RST_STREAM with an error code of NO_ERROR after sending a complete response and closing its stream. Clients **MUST NOT** discard responses as a result of receiving such a RST_STREAM, though clients can always discard responses at their discretion for other reasons.

4.2.1. Header Compression

HTTP/QUIC uses HPACK header compression as described in [RFC7541]. HPACK was designed for HTTP/2 with the assumption of in-order delivery such as that provided by TCP. A sequence of encoded header

blocks must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

QUIC streams provide in-order delivery of data sent on those streams, but there are no guarantees about order of delivery between streams. To achieve in-order delivery of HEADERS frames in QUIC, the HPACK-bearing frames contain a counter which can be used to ensure in-order processing. Data (request/response bodies) which arrive out of order are buffered until the corresponding HEADERS arrive.

This does introduce head-of-line blocking: if the packet containing HEADERS for stream N is lost or reordered then the HEADERS for stream N+4 cannot be processed until it has been retransmitted successfully, even though the HEADERS for stream N+4 may have arrived.

DISCUSS: Keep HPACK with HOLB? Redesign HPACK to be order-invariant? How much do we need to retain compatibility with HTTP/2's HPACK?

4.2.2. The CONNECT Method

The pseudo-method CONNECT ([RFC7231], Section 4.3.6) is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources. In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes.

A CONNECT request in HTTP/QUIC functions in the same manner as in HTTP/2. The request MUST be formatted as described in [RFC7540], Section 8.3. A CONNECT request that does not conform to these restrictions is malformed. The message data stream MUST NOT be closed at the end of the request.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6, on the message control stream.

All QUIC STREAM frames on the message data stream correspond to data sent on the TCP connection. Any QUIC STREAM frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is written to the data stream by the proxy. Note that the

size and number of TCP segments is not guaranteed to map predictably to the size and number of QUIC STREAM frames.

The TCP connection can be closed by either peer. When the client half-closes the data stream, the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will half-close the corresponding data stream. TCP connections which remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT half-close connections on which they are still expecting data.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type HTTP_CONNECT_ERROR (Section 6.1). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the QUIC connection.

4.3. Stream Priorities

HTTP/QUIC uses the priority scheme described in [RFC7540] Section 5.3. In this priority scheme, a given stream can be designated as dependent upon another stream, which expresses the preference that the latter stream (the "parent" stream) be allocated resources before the former stream (the "dependent" stream). Taken together, the dependencies across all streams in a connection form a dependency tree. The structure of the dependency tree changes as PRIORITY frames add, remove, or change the dependency links between streams.

For consistency's sake, all PRIORITY frames MUST refer to the message control stream of the dependent request, not the data stream.

4.4. Server Push

HTTP/QUIC supports server push as described in [RFC7540]. During connection establishment, the client indicates whether it is willing to receive server pushes via the SETTINGS_DISABLE_PUSH setting in the SETTINGS frame (see Section 3), which defaults to 1 (true).

As with server push for HTTP/2, the server initiates a server push by sending a PUSH_PROMISE frame containing the StreamID of the stream to be pushed, as well as request header fields attributed to the request. The PUSH_PROMISE frame is sent on the control stream of the associated (client-initiated) request, while the Promised Stream ID field specifies the Stream ID of the control stream for the server-initiated request.

The server push response is conveyed in the same way as a non-server-push response, with response headers and (if present) trailers carried by HEADERS frames sent on the control stream, and response body (if any) sent via the corresponding data stream.

5. HTTP Framing Layer

Frames are used only on the connection (stream 3) and message (streams 5, 9, etc.) control streams. Other streams carry data payload and are not framed at the HTTP layer.

This section describes HTTP framing in QUIC and highlights some differences from HTTP/2 framing. For more detail on differences from HTTP/2, see Section 7.1.

5.1. Frame Layout

All frames have the following format:

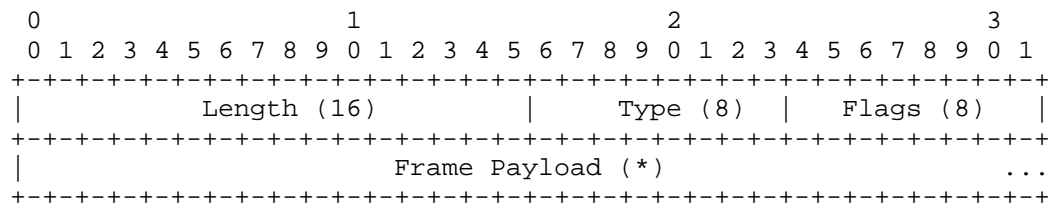


Figure 1: HTTP/QUIC frame format

5.2. Frame Definitions

5.2.1. HEADERS

The HEADERS frame (type=0x1) is used to carry part of a header set, compressed using HPACK [RFC7541].

One flag is defined:

End Header Block (0x4): This frame concludes a header block.

A HEADERS frame with any other flags set MUST be treated as a connection error of type HTTP_MALFORMED_HEADERS.

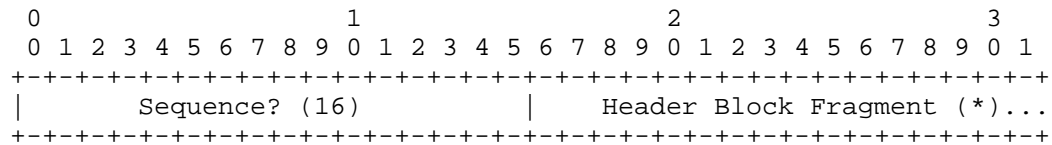


Figure 2: HEADERS frame payload

The HEADERS frame payload has the following fields:

Sequence Number: Present only on the first frame of a header block sequence. This MUST be set to zero on the first header block sequence, and incremented on each header block.

The next frame on the same stream after a HEADERS frame without the EHB flag set MUST be another HEADERS frame. A receiver MUST treat the receipt of any other type of frame as a stream error of type HTTP_INTERRUPTED_HEADERS. (Note that QUIC can intersperse data from other streams between frames, or even during transmission of frames, so multiplexing is not blocked by this requirement.)

A full header block is contained in a sequence of zero or more HEADERS frames without EHB set, followed by a HEADERS frame with EHB set.

On receipt, header blocks (HEADERS, PUSH_PROMISE) MUST be processed by the HPACK decoder in sequence. If a block is missing, all subsequent HPACK frames MUST be held until it arrives, or the connection terminated.

When the Sequence counter reaches its maximum value (0xFFFF), the next increment returns it to zero. An endpoint MUST NOT wrap the Sequence counter to zero until the previous zero-value header block has been confirmed received.

5.2.2. PRIORITY

The PRIORITY (type=0x02) frame specifies the sender-advised priority of a stream and is substantially different from [RFC7540]. In order to support ordering, it MUST be sent only on the connection control stream. The format has been modified to accommodate not being sent on-stream and the larger stream ID space of QUIC.

The semantics of the Stream Dependency, Weight, and E flag are the same as in HTTP/2.

The flags defined are:

E (0x01): Indicates that the stream dependency is exclusive (see [RFC7540] Section 5.3).

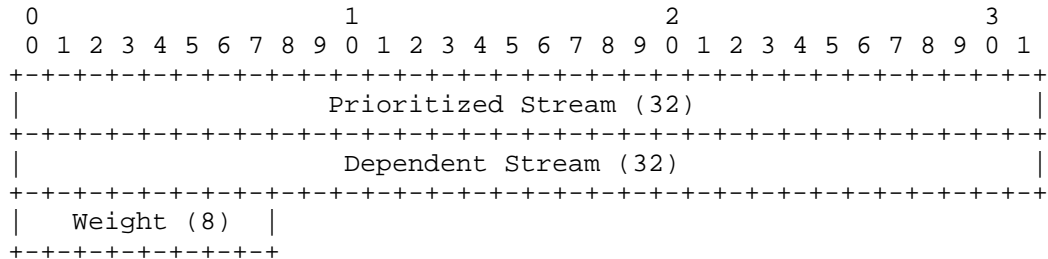


Figure 3: PRIORITY frame payload

The HEADERS frame payload has the following fields:

Prioritized Stream: A 32-bit stream identifier for the message control stream whose priority is being updated.

Stream Dependency: A 32-bit stream identifier for the stream that this stream depends on (see Section 4.3 and [RFC7540] Section 5.3).

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see [RFC7540] Section 5.3). Add one to the value to obtain a weight between 1 and 256.

A PRIORITY frame MUST have a payload length of nine octets. A PRIORITY frame of any other length MUST be treated as a connection error of type HTTP_MALFORMED_PRIORITY.

5.2.3. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior, and is substantially different from [RFC7540]. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - a peer uses SETTINGS to advertise a set of supported values. The recipient can then choose which entries from this list are also acceptable and proceed with the value it has chosen. (This choice could be announced in a field of an extension frame, or in its own value in SETTINGS.)

Different values for the same parameter can be advertised by each peer. For example, a client might permit a very large HPACK state table while a server chooses to use a small one to conserve memory.

Parameters **MUST NOT** occur more than once. A receiver **MAY** treat the presence of the same parameter more than once as a connection error of type `HTTP_MALFORMED_SETTINGS`.

The `SETTINGS` frame defines no flags.

The payload of a `SETTINGS` frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and a length-prefixed binary value.

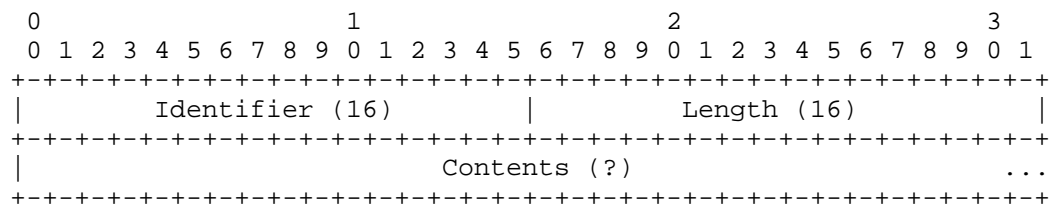


Figure 4: `SETTINGS` value format

A zero-length content indicates that the setting value is a Boolean and true. False is indicated by the absence of the setting.

Non-zero-length values **MUST** be compared against the remaining length of the `SETTINGS` frame. Any value which purports to cross the end of the frame **MUST** cause the `SETTINGS` frame to be considered malformed and trigger a connection error of type `HTTP_MALFORMED_SETTINGS`.

An implementation **MUST** ignore the contents for any `SETTINGS` identifier it does not understand.

`SETTINGS` frames always apply to a connection, never a single stream. A `SETTINGS` frame **MUST** be sent as the first frame of the connection control stream (see Section 4) by each peer, and **MUST NOT** be sent subsequently or on any other stream. If an endpoint receives an `SETTINGS` frame on a different stream, the endpoint **MUST** respond with a connection error of type `HTTP_SETTINGS_ON_WRONG_STREAM`. If an endpoint receives a second `SETTINGS` frame, the endpoint **MUST** respond with a connection error of type `HTTP_MULTIPLE_SETTINGS`.

The `SETTINGS` frame affects connection state. A badly formed or incomplete `SETTINGS` frame **MUST** be treated as a connection error (Section 5.4.1) of type `HTTP_MALFORMED_SETTINGS`.

5.2.3.1. Integer encoding

Settings which are integers are transmitted in network byte order. Leading zero octets are permitted, but implementations SHOULD use only as many bytes as are needed to represent the value. An integer MUST NOT be represented in more bytes than would be used to transfer the maximum permitted value.

5.2.3.2. Defined SETTINGS Parameters

The following settings are defined in HTTP/QUIC:

SETTINGS_HEADER_TABLE_SIZE (0x1): An integer with a maximum value of $2^{32} - 1$.

SETTINGS_DISABLE_PUSH (0x2): Transmitted as a Boolean; replaces SETTINGS_ENABLE_PUSH

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): An integer with a maximum value of $2^{32} - 1$.

5.2.3.3. Usage in 0-RTT

When a 0-RTT QUIC connection is being used, the client's initial requests will be sent before the arrival of the server's SETTINGS frame. Clients SHOULD cache at least the following settings about servers:

- o SETTINGS_HEADER_TABLE_SIZE
- o SETTINGS_MAX_HEADER_LIST_SIZE

Clients MUST comply with cached settings until the server's current settings are received. If a client does not have cached values, it SHOULD assume the following values:

- o SETTINGS_HEADER_TABLE_SIZE: 0 octets
- o SETTINGS_MAX_HEADER_LIST_SIZE: 16,384 octets

Servers MAY continue processing data from clients which exceed its current configuration during the initial flight. In this case, the client MUST apply the new settings immediately upon receipt.

If the connection is closed because these or other constraints were violated during the 0-RTT flight (e.g. with HTTP_HPACK_DECOMPRESSION_FAILED), clients MAY establish a new connection and retry any 0-RTT requests using the settings sent by

the server on the closed connection. (This assumes that only requests that are safe to retry are sent in 0-RTT.) If the connection was closed before the SETTINGS frame was received, clients SHOULD discard any cached values and use the defaults above on the next connection.

5.2.4. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x05) is used to carry a request header set from server to client, as in HTTP/2. It defines no flags.

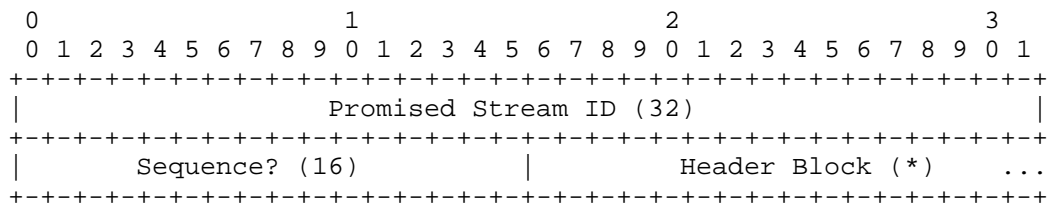


Figure 5: PUSH_PROMISE frame payload

The payload consists of:

Promised Stream ID: A 32-bit Stream ID indicating the QUIC stream on which the response headers will be sent. (The response body stream is implied by the headers stream, as defined in Section 4.)

HPACK Sequence: A sixteen-bit counter, equivalent to the Sequence field in HEADERS

Payload: HPACK-compressed request headers for the promised response.

6. Error Handling

QUIC allows the application to abruptly terminate individual streams or the entire connection when an error is encountered. These are referred to as "stream errors" or "connection errors" and are described in more detail in [QUIC-TRANSPORT].

HTTP/QUIC requires that only data streams be terminated abruptly. Terminating a message control stream will result in an error of type HTTP_RST_CONTROL_STREAM.

This section describes HTTP-specific error codes which can be used to express the cause of a connection or stream error.

6.1. HTTP-Defined QUIC Error Codes

QUIC allocates error codes 0x0000-0x3FFF to application protocol definition. The following error codes are defined by HTTP for use in QUIC RST_STREAM, GOAWAY, and CONNECTION_CLOSE frames.

HTTP_PUSH_REFUSED (0x01): The server has attempted to push content which the client will not accept on this connection.

HTTP_INTERNAL_ERROR (0x02): An internal error has occurred in the HTTP stack.

HTTP_PUSH_ALREADY_IN_CACHE (0x03): The server has attempted to push content which the client has cached.

HTTP_REQUEST_CANCELLED (0x04): The client no longer needs the requested data.

HTTP_HPACK_DECOMPRESSION_FAILED (0x05): HPACK failed to decompress a frame and cannot continue.

HTTP_CONNECT_ERROR (0x06): The connection established in response to a CONNECT request was reset or abnormally closed.

HTTP_EXCESSIVE_LOAD (0x07): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

HTTP_VERSION_FALLBACK (0x08): The requested operation cannot be served over HTTP/QUIC. The peer should retry over HTTP/2.

HTTP_MALFORMED_HEADERS (0x09): A HEADERS frame has been received with an invalid format.

HTTP_MALFORMED_PRIORITY (0x0A): A PRIORITY frame has been received with an invalid format.

HTTP_MALFORMED_SETTINGS (0x0B): A SETTINGS frame has been received with an invalid format.

HTTP_MALFORMED_PUSH_PROMISE (0x0C): A PUSH_PROMISE frame has been received with an invalid format.

HTTP_INTERRUPTED_HEADERS (0x0E): A HEADERS frame without the End Header Block flag was followed by a frame other than HEADERS.

HTTP_SETTINGS_ON_WRONG_STREAM (0x0F): A SETTINGS frame was received on a request control stream.

HTTP_MULTIPLE_SETTINGS (0x10): More than one SETTINGS frame was received.

HTTP_RST_CONTROL_STREAM (0x11): A message control stream closed abruptly.

7. Considerations for Transitioning from HTTP/2

HTTP/QUIC is strongly informed by HTTP/2, and bears many similarities. This section points out important differences from HTTP/2 and describes how to map HTTP/2 extensions into HTTP/QUIC.

7.1. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided away on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required.

Frame payloads are largely drawn from [RFC7540]. However, QUIC includes many features (e.g. flow control) which are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/QUIC. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/QUIC implementations. However, even equivalent frames between the two mappings are not identical.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/QUIC will break them.

For example, implicit in the HTTP/2 prioritization scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of priority changes in HTTP/QUIC, PRIORITY frames are sent on the connection control stream and the PRIORITY section is removed from the HEADERS frame.

Other than this issue, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with Stream 3 in HTTP/QUIC.

Below is a listing of how each HTTP/2 frame type is mapped:

DATA (0x0): Instead of DATA frames, HTTP/QUIC uses a separate data stream. See Section 4.

HEADERS (0x1): As described above, the PRIORITY region of HEADERS is not supported. A separate PRIORITY frame MUST be used. Padding is not defined in HTTP/QUIC frames. See Section 5.2.1.

PRIORITY (0x2): As described above, the PRIORITY frame is sent on the connection control stream. See Section 5.2.2.

RST_STREAM (0x3): RST_STREAM frames do not exist, since QUIC provides stream lifecycle management.

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 5.2.3 and Section 7.2.

PUSH_PROMISE (0x5): See Section 5.2.4.

PING (0x6): PING frames do not exist, since QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY frames do not exist, since QUIC provides equivalent functionality.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted, and HEADERS frames can be used in series.

The IANA registry of frame types has been updated in Section 9.3 to include references to the definition for each frame type in HTTP/2 and in HTTP/QUIC. Frames not defined as available in HTTP/QUIC SHOULD NOT be sent and SHOULD be ignored as unknown on receipt.

7.2. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, at the beginning of the connection, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/QUIC. The HTTP-level options that are retained in HTTP/QUIC have the same value as in HTTP/2.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE: See Section 5.2.3.2.

SETTINGS_ENABLE_PUSH: See SETTINGS_DISABLE_PUSH in Section 5.2.3.2.

SETTINGS_MAX_CONCURRENT_STREAMS: QUIC requires the maximum number of incoming streams per connection to be specified in the initial transport handshake. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/QUIC. Specifying it in the SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: See Section 5.2.3.2.

Settings defined by extensions to HTTP/2 MAY be expressed as integers with a maximum value of $2^{32}-1$, if they are applicable to HTTP/QUIC, but SHOULD have a specification describing their usage. Fields for this purpose have been added to the IANA registry in Section 9.4.

7.3. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, because the error code space is shared between multiple components, there is no direct portability of HTTP/2 error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to QUIC error codes as follows:

NO_ERROR (0x0): QUIC_NO_ERROR

PROTOCOL_ERROR (0x1): No single mapping. See new HTTP_MALFORMED_* error codes defined in Section 6.1.

INTERNAL_ERROR (0x2) HTTP_INTERNAL_ERROR in Section 6.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA from the QUIC layer.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION from the QUIC layer.

FRAME_SIZE_ERROR (0x6) No single mapping. See new error codes defined in Section 6.1.

REFUSED_STREAM (0x7): Not applicable, since QUIC handles stream management. Would provoke a QUIC_TOO_MANY_OPEN_STREAMS from the QUIC layer.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 6.1.

COMPRESSION_ERROR (0x9): HTTP_HPACK_DECOMPRESSION_FAILED in Section 6.1.

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 6.1.

ENHANCE_YOUR_CALM (0xb): HTTP_EXCESSIVE_LOAD in Section 6.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): HTTP_VERSION_FALLBACK in Section 6.1.

Error codes defined by HTTP/2 extensions need to be re-registered for HTTP/QUIC if still applicable. See Section 9.5.

8. Security Considerations

The security considerations of HTTP over QUIC should be comparable to those of HTTP/2.

The modified SETTINGS format contains nested length elements, which could pose a security risk to an uncautious implementer. A SETTINGS frame parser MUST ensure that the length of the frame exactly matches the length of the settings it contains.

9. IANA Considerations

9.1. Registration of HTTP/QUIC Identification String

This document creates a new registration for the identification of HTTP/QUIC in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "hq" string identifies HTTP/QUIC:

Protocol: HTTP over QUIC

Identification Sequence: 0x68 0x71 ("hq")

Specification: This document

9.2. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter" registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 2.1

9.3. Existing Frame Types

This document adds two new columns to the "HTTP/2 Frame Type" registry defined in [RFC7540]:

Supported Protocols: Indicates which associated protocols use the frame type. Values MUST be one of:

- * "HTTP/2 only"
- * "HTTP/QUIC only"
- * "Both"

HTTP/QUIC Specification: Indicates where this frame's behavior over QUIC is defined; required if the frame is supported over QUIC.

Values for existing registrations are assigned by this document:

Frame Type	Supported Protocols	HTTP/QUIC Specification
DATA	HTTP/2 only	N/A
HEADERS	Both	Section 5.2.1
PRIORITY	Both	Section 5.2.2
RST_STREAM	HTTP/2 only	N/A
SETTINGS	Both	Section 5.2.3
PUSH_PROMISE	Both	Section 5.2.4
PING	HTTP/2 only	N/A
GOAWAY	HTTP/2 only	N/A
WINDOW_UPDATE	HTTP/2 only	N/A
CONTINUATION	HTTP/2 only	N/A

The "Specification" column is renamed to "HTTP/2 specification" and is only required if the frame is supported over HTTP/2.

9.4. Settings Parameters

This document adds two new columns to the "HTTP/2 Settings" registry defined in [RFC7540]:

Supported Protocols: Indicates which associated protocols use the setting. Values MUST be one of:

- * "HTTP/2 only"
- * "HTTP/QUIC only"
- * "Both"

HTTP/QUIC Specification: Indicates where this setting's behavior over QUIC is defined; required if the frame is supported over QUIC.

Values for existing registrations are assigned by this document:

Setting Name	Supported Protocols	HTTP/QUIC Specification
HEADER_TABLE_SIZE	Both	Section 5.2.3.2
ENABLE_PUSH / DISABLE_PUSH	Both	Section 5.2.3.2
MAX_CONCURRENT_STREAMS	HTTP/2 Only	N/A
INITIAL_WINDOW_SIZE	HTTP/2 Only	N/A
MAX_FRAME_SIZE	HTTP/2 Only	N/A
MAX_HEADER_LIST_SIZE	Both	Section 5.2.3.2

The "Specification" column is renamed to "HTTP/2 Specification" and is only required if the setting is supported over HTTP/2.

9.5. Error Codes

This document establishes a registry for HTTP/QUIC error codes. The "HTTP/QUIC Error Code" registry manages a 30-bit space. The "HTTP/QUIC Error Code" registry operates under the "Expert Review" policy [RFC5226].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is optional.

Code: The 30-bit error code value.

Description: A brief description of the error code semantics, longer if no detailed specification is provided.

Specification: An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

Name	Code	Description	Specification
HTTP_PUSH_REFUSED	0x01	Client refused pushed content	Section 6.1
HTTP_INTERNAL_ERROR	0x02	Internal error	Section 6.1
HTTP_PUSH_ALREADY_IN_CACHE	0x03	Pushed content already cached	Section 6.1
HTTP_REQUEST_CANCELLED	0x04	Data no longer needed	Section 6.1
HTTP_HPACK_DECOMPRESSION_FAILED	0x05	HPACK cannot continue	Section 6.1
HTTP_CONNECT_ERROR	0x06	TCP reset or error on CONNECT request	Section 6.1
HTTP_EXCESSIVE_LOAD	0x07	Peer generating excessive load	Section 6.1
HTTP_VERSION_FALLBACK	0x08	Retry over HTTP/2	Section 6.1
HTTP_MALFORMED_HEADERS	0x09	Invalid HEADERS frame	Section 6.1
HTTP_MALFORMED_PRIORITY	0x0A	Invalid PRIORITY frame	Section 6.1
HTTP_MALFORMED_SETTINGS	0x0B	Invalid SETTINGS frame	Section 6.1

HTTP_MALFORMED_PUSH_PROMISE	0x0 C	Invalid PUSH_PROMISE frame	Section 6.1
HTTP_INTERRUPTED_HEADERS	0x0 E	Incomplete HEADERS block	Section 6.1
HTTP_SETTINGS_ON_WRONG_STREAM	0x0 F	SETTINGS frame on a request control stream	Section 6.1
HTTP_MULTIPLE_SETTINGS	0x1 0	Multiple SETTINGS frames	Section 6.1
HTTP_RST_CONTROL_STREAM	0x1 1	Message control stream was RST	Section 6.1

10. References

10.1. Normative References

- [QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<http://www.rfc-editor.org/info/rfc793>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<http://www.rfc-editor.org/info/rfc7230>>.

- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<http://www.rfc-editor.org/info/rfc7231>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<http://www.rfc-editor.org/info/rfc7540>>.
- [RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<http://www.rfc-editor.org/info/rfc7541>>.
- [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<http://www.rfc-editor.org/info/rfc7838>>.

10.2. Informative References

- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.

Appendix A. Contributors

The original authors of this specification were Robbie Shade and Mike Warren.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-http-01:

o SETTINGS changes (#181):

- * SETTINGS can be sent only once at the start of a connection; no changes thereafter
- * SETTINGS_ACK removed

- * Settings can only occur in the SETTINGS frame a single time
 - * Boolean format updated
 - o Alt-Svc parameter changed from "v" to "quic"; format updated (#229)
 - o Closing the connection control stream or any message control stream is a fatal error (#176)
 - o HPACK Sequence counter can wrap (#173)
 - o 0-RTT guidance added
 - o Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)
- B.2. Since draft-ietf-quic-http-00:
- o Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)
 - o Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)
 - o Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01
 - o Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)
 - o Described CONNECT pseudo-method (#95)
 - o Updated ALPN token and Alt-Svc guidance (#13,#87)
 - o Application-layer-defined error codes (#19,#74)
- B.3. Since draft-shade-quic-http2-mapping-00:
- o Adopted as base for draft-ietf-quic-http.
 - o Updated authors/editors list.

Author's Address

Mike Bishop (editor)
Microsoft

Email: Michael.Bishop@microsoft.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 6 August 2021

M. Bishop, Ed.
Akamai
2 February 2021

Hypertext Transfer Protocol Version 3 (HTTP/3)
draft-ietf-quic-http-34

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to HTTP/3.

DO NOT DEPLOY THIS VERSION OF HTTP

DO NOT DEPLOY THIS VERSION OF HTTP/3 UNTIL IT IS IN AN RFC. This version is still a work in progress. For trial deployments, please use earlier versions.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-http>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 6 August 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Prior versions of HTTP	5
1.2. Delegation to QUIC	5
2. HTTP/3 Protocol Overview	5
2.1. Document Organization	6
2.2. Conventions and Terminology	7
3. Connection Setup and Management	8
3.1. Discovering an HTTP/3 Endpoint	8
3.1.1. HTTP Alternative Services	9
3.1.2. Other Schemes	10
3.2. Connection Establishment	10
3.3. Connection Reuse	11
4. HTTP Request Lifecycle	12
4.1. HTTP Message Exchanges	12
4.1.1. Field Formatting and Compression	14
4.1.2. Request Cancellation and Rejection	17
4.1.3. Malformed Requests and Responses	18
4.2. The CONNECT Method	19
4.3. HTTP Upgrade	21
4.4. Server Push	21
5. Connection Closure	23
5.1. Idle Connections	23
5.2. Connection Shutdown	24
5.3. Immediate Application Closure	26
5.4. Transport Closure	26
6. Stream Mapping and Usage	27
6.1. Bidirectional Streams	27
6.2. Unidirectional Streams	28
6.2.1. Control Streams	29
6.2.2. Push Streams	30

6.2.3. Reserved Stream Types	30
7. HTTP Framing Layer	31
7.1. Frame Layout	32
7.2. Frame Definitions	32
7.2.1. DATA	32
7.2.2. HEADERS	33
7.2.3. CANCEL_PUSH	33
7.2.4. SETTINGS	35
7.2.5. PUSH_PROMISE	38
7.2.6. GOAWAY	39
7.2.7. MAX_PUSH_ID	40
7.2.8. Reserved Frame Types	41
8. Error Handling	41
8.1. HTTP/3 Error Codes	42
9. Extensions to HTTP/3	43
10. Security Considerations	44
10.1. Server Authority	44
10.2. Cross-Protocol Attacks	44
10.3. Intermediary Encapsulation Attacks	45
10.4. Cacheability of Pushed Responses	45
10.5. Denial-of-Service Considerations	45
10.5.1. Limits on Field Section Size	46
10.5.2. CONNECT Issues	47
10.6. Use of Compression	47
10.7. Padding and Traffic Analysis	48
10.8. Frame Parsing	48
10.9. Early Data	49
10.10. Migration	49
10.11. Privacy Considerations	49
11. IANA Considerations	49
11.1. Registration of HTTP/3 Identification String	50
11.2. New Registries	50
11.2.1. Frame Types	50
11.2.2. Settings Parameters	52
11.2.3. Error Codes	53
11.2.4. Stream Types	55
12. References	56
12.1. Normative References	56
12.2. Informative References	57
Appendix A. Considerations for Transitioning from HTTP/2	58
A.1. Streams	59
A.2. HTTP Frame Types	60
A.2.1. Prioritization Differences	60
A.2.2. Field Compression Differences	60
A.2.3. Flow Control Differences	61
A.2.4. Guidance for New Frame Type Definitions	61
A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types	61
A.3. HTTP/2 SETTINGS Parameters	62

A.4. HTTP/2 Error Codes	64
A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors	65
Appendix B. Change Log	65
B.1. Since draft-ietf-quic-http-32	65
B.2. Since draft-ietf-quic-http-31	66
B.3. Since draft-ietf-quic-http-30	66
B.4. Since draft-ietf-quic-http-29	66
B.5. Since draft-ietf-quic-http-28	66
B.6. Since draft-ietf-quic-http-27	66
B.7. Since draft-ietf-quic-http-26	66
B.8. Since draft-ietf-quic-http-25	66
B.9. Since draft-ietf-quic-http-24	67
B.10. Since draft-ietf-quic-http-23	67
B.11. Since draft-ietf-quic-http-22	67
B.12. Since draft-ietf-quic-http-21	68
B.13. Since draft-ietf-quic-http-20	68
B.14. Since draft-ietf-quic-http-19	69
B.15. Since draft-ietf-quic-http-18	69
B.16. Since draft-ietf-quic-http-17	69
B.17. Since draft-ietf-quic-http-16	70
B.18. Since draft-ietf-quic-http-15	70
B.19. Since draft-ietf-quic-http-14	70
B.20. Since draft-ietf-quic-http-13	70
B.21. Since draft-ietf-quic-http-12	71
B.22. Since draft-ietf-quic-http-11	71
B.23. Since draft-ietf-quic-http-10	71
B.24. Since draft-ietf-quic-http-09	71
B.25. Since draft-ietf-quic-http-08	72
B.26. Since draft-ietf-quic-http-07	72
B.27. Since draft-ietf-quic-http-06	72
B.28. Since draft-ietf-quic-http-05	72
B.29. Since draft-ietf-quic-http-04	72
B.30. Since draft-ietf-quic-http-03	72
B.31. Since draft-ietf-quic-http-02	73
B.32. Since draft-ietf-quic-http-01	73
B.33. Since draft-ietf-quic-http-00	73
B.34. Since draft-shade-quic-http2-mapping-00	74
Acknowledgments	74
Author's Address	75

1. Introduction

HTTP semantics ([SEMANTICS]) are used for a broad range of services on the Internet. These semantics have most commonly been used with HTTP/1.1 and HTTP/2. HTTP/1.1 has been used over a variety of transport and session layers, while HTTP/2 has been used primarily with TLS over TCP. HTTP/3 supports the same semantics over a new transport protocol, QUIC.

1.1. Prior versions of HTTP

HTTP/1.1 ([HTTP11]) uses whitespace-delimited text fields to convey HTTP messages. While these exchanges are human-readable, using whitespace for message formatting leads to parsing complexity and excessive tolerance of variant behavior.

Because HTTP/1.1 does not include a multiplexing layer, multiple TCP connections are often used to service requests in parallel. However, that has a negative impact on congestion control and network efficiency, since TCP does not share congestion control across multiple connections.

HTTP/2 ([HTTP2]) introduced a binary framing and multiplexing layer to improve latency without modifying the transport layer. However, because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was directly impacted by the lost packet.

1.2. Delegation to QUIC

The QUIC transport protocol incorporates stream multiplexing and per-stream flow control, similar to that provided by the HTTP/2 framing layer. By providing reliability at the stream level and congestion control across the entire connection, QUIC has the capability to improve the performance of HTTP compared to a TCP mapping. QUIC also incorporates TLS 1.3 ([TLS13]) at the transport layer, offering comparable confidentiality and integrity to running TLS over TCP, with the improved connection setup latency of TCP Fast Open ([TFO]).

This document defines HTTP/3, a mapping of HTTP semantics over the QUIC transport protocol, drawing heavily on the design of HTTP/2. HTTP/3 relies on QUIC to provide confidentiality and integrity protection of data; peer authentication; and reliable, in-order, per-stream delivery. While delegating stream lifetime and flow control issues to QUIC, a binary framing similar to the HTTP/2 framing is used on each stream. Some HTTP/2 features are subsumed by QUIC, while other features are implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [HTTP2].

2. HTTP/3 Protocol Overview

HTTP/3 provides a transport for HTTP semantics using the QUIC transport protocol and an internal framing layer similar to HTTP/2.

Once a client knows that an HTTP/3 server exists at a certain endpoint, it opens a QUIC connection. QUIC provides protocol negotiation, stream-based multiplexing, and flow control. Discovery of an HTTP/3 endpoint is described in Section 3.1.

Within each stream, the basic unit of HTTP/3 communication is a frame (Section 7.2). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 4.1). Frames that apply to the entire connection are conveyed on a dedicated control stream.

Multiplexing of requests is performed using the QUIC stream abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

Server push is an interaction mode introduced in HTTP/2 ([HTTP2]) that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. Several HTTP/3 frames are used to manage server push, such as PUSH_PROMISE, MAX_PUSH_ID, and CANCEL_PUSH.

As in HTTP/2, request and response fields are compressed for transmission. Because HPACK ([HPACK]) relies on in-order transmission of compressed field sections (a guarantee not provided by QUIC), HTTP/3 replaces HPACK with QPACK ([QPACK]). QPACK uses separate unidirectional streams to modify and track field table state, while encoded field sections refer to the state of the table without modifying it.

2.1. Document Organization

The following sections provide a detailed overview of the lifecycle of an HTTP/3 connection:

- * Connection Setup and Management (Section 3) covers how an HTTP/3 endpoint is discovered and an HTTP/3 connection is established.
- * HTTP Request Lifecycle (Section 4) describes how HTTP semantics are expressed using frames.
- * Connection Closure (Section 5) describes how HTTP/3 connections are terminated, either gracefully or abruptly.

The details of the wire protocol and interactions with the transport are described in subsequent sections:

- * Stream Mapping and Usage (Section 6) describes the way QUIC streams are used.
- * HTTP Framing Layer (Section 7) describes the frames used on most streams.
- * Error Handling (Section 8) describes how error conditions are handled and expressed, either on a particular stream or for the connection as a whole.

Additional resources are provided in the final sections:

- * Extensions to HTTP/3 (Section 9) describes how new capabilities can be added in future documents.
- * A more detailed comparison between HTTP/2 and HTTP/3 can be found in Appendix A.

2.2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].

The following terms are used:

abort: An abrupt termination of a connection or stream, possibly due to an error condition.

client: The endpoint that initiates an HTTP/3 connection. Clients send HTTP requests and receive HTTP responses.

connection: A transport-layer connection between two endpoints, using QUIC as the transport protocol.

connection error: An error that affects the entire HTTP/3 connection.

endpoint: Either the client or server of the connection.

frame: The smallest unit of communication on a stream in HTTP/3, consisting of a header and a variable-length sequence of bytes structured according to the frame type.

Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC CONNECTION_CLOSE frames." References without this preface refer to frames defined in Section 7.2.

HTTP/3 connection: A QUIC connection where the negotiated application protocol is HTTP/3.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

receiver: An endpoint that is receiving frames.

sender: An endpoint that is transmitting frames.

server: The endpoint that accepts an HTTP/3 connection. Servers receive HTTP requests and send HTTP responses.

stream: A bidirectional or unidirectional bytestream provided by the QUIC transport. All streams within an HTTP/3 connection can be considered "HTTP/3 streams," but multiple stream types are defined within HTTP/3.

stream error: An application-level error on the individual stream.

The term "content" is defined in Section 6.4 of [SEMANTICS].

Finally, the terms "resource", "message", "user agent", "origin server", "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 3 of [SEMANTICS].

Packet diagrams in this document use the format defined in Section 1.3 of [QUIC-TRANSPORT] to illustrate the order and size of fields.

3. Connection Setup and Management

3.1. Discovering an HTTP/3 Endpoint

HTTP relies on the notion of an authoritative response: a response that has been determined to be the most appropriate response for that request given the state of the target resource at the time of response message origination by (or at the direction of) the origin server identified within the target URI. Locating an authoritative server for an HTTP URI is discussed in Section 4.3 of [SEMANTICS].

The "https" scheme associates authority with possession of a certificate that the client considers to be trustworthy for the host identified by the authority component of the URI. Upon receiving a server certificate in the TLS handshake, the client **MUST** verify that the certificate is an acceptable match for the URI's origin server using the process described in Section 4.3.4 of [SEMANTICS]. If the certificate cannot be verified with respect to the URI's origin server, the client **MUST NOT** consider the server authoritative for that origin.

A client **MAY** attempt access to a resource with an "https" URI by resolving the host identifier to an IP address, establishing a QUIC connection to that address on the indicated port (including validation of the server certificate as described above), and sending an HTTP/3 request message targeting the URI to the server over that secured connection. Unless some other mechanism is used to select HTTP/3, the token "h3" is used in the Application Layer Protocol Negotiation (ALPN; see [RFC7301]) extension during the TLS handshake.

Connectivity problems (e.g., blocking UDP) can result in QUIC connection establishment failure; clients **SHOULD** attempt to use TCP-based versions of HTTP in this case.

Servers **MAY** serve HTTP/3 on any UDP port; an alternative service advertisement always includes an explicit port, and URIs contain either an explicit port or a default port associated with the scheme.

3.1.1. HTTP Alternative Services

An HTTP origin can advertise the availability of an equivalent HTTP/3 endpoint via the Alt-Svc HTTP response header field or the HTTP/2 ALTSVC frame ([ALTSVC]), using the "h3" ALPN token.

For example, an origin could indicate in an HTTP response that HTTP/3 was available on UDP port 50781 at the same hostname by including the following header field:

```
Alt-Svc: h3=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/3 support, a client **MAY** attempt to establish a QUIC connection to the indicated host and port; if this connection is successful, the client can send HTTP requests using the mapping described in this document.

3.1.2. Other Schemes

Although HTTP is independent of the transport protocol, the "http" scheme associates authority with the ability to receive TCP connections on the indicated port of whatever host is identified within the authority component. Because HTTP/3 does not use TCP, HTTP/3 cannot be used for direct access to the authoritative server for a resource identified by an "http" URI. However, protocol extensions such as [ALTSVC] permit the authoritative server to identify other services that are also authoritative and that might be reachable over HTTP/3.

Prior to making requests for an origin whose scheme is not "https", the client **MUST** ensure the server is willing to serve that scheme. For origins whose scheme is "http", an experimental method to accomplish this is described in [RFC8164]. Other mechanisms might be defined for various schemes in the future.

3.2. Connection Establishment

HTTP/3 relies on QUIC version 1 as the underlying transport. The use of other QUIC transport versions with HTTP/3 **MAY** be defined by future specifications.

QUIC version 1 uses TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients **MUST** support a mechanism to indicate the target host to the server during the TLS handshake. If the server is identified by a domain name ([DNS-TERMS]), clients **MUST** send the Server Name Indication (SNI; [RFC6066]) TLS extension unless an alternative mechanism to indicate the target host is used.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/3 support is indicated by selecting the ALPN token "h3" in the TLS handshake. Support for other application-layer protocols **MAY** be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP/3-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 7.2.4) **MUST** be sent by each endpoint as the initial frame of their respective HTTP control stream; see Section 6.2.1.

3.3. Connection Reuse

HTTP/3 connections are persistent across multiple requests. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. To use an existing connection for a new origin, clients MUST validate the certificate presented by the server for the new origin server using the process described in Section 4.3.4 of [SEMANTICS]. This implies that clients will need to retain the server certificate and any additional information needed to verify that certificate; clients which do not do so will be unable to reuse the connection for additional origins.

If the certificate is not acceptable with regard to the new origin for any reason, the connection MUST NOT be reused and a new connection SHOULD be established for the new origin. If the reason the certificate cannot be verified might apply to other origins already associated with the connection, the client SHOULD re-validate the server certificate for those origins. For instance, if validation of a certificate fails because the certificate has expired or been revoked, this might be used to invalidate all other origins for which that certificate was used to establish authority.

Clients SHOULD NOT open more than one HTTP/3 connection to a given IP address and UDP port, where the IP address and port might be derived from a URI, a selected alternative service ([ALTSVC]), a configured proxy, or name resolution of any of these. A client MAY open multiple HTTP/3 connections to the same IP address and UDP port using different transport or TLS configurations but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open HTTP/3 connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the HTTP/3 connection, the terminating endpoint SHOULD first send a GOAWAY frame (Section 5.2) so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

A server that does not wish clients to reuse HTTP/3 connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request; see Section 7.4 of [SEMANTICS].

4. HTTP Request Lifecycle

4.1. HTTP Message Exchanges

A client sends an HTTP request on a request stream, which is a client-initiated bidirectional QUIC stream; see Section 6.1. A client **MUST** send only a single request on a given stream. A server sends zero or more interim HTTP responses on the same stream as the request, followed by a single final HTTP response, as detailed below. See Section 15 of [SEMANTICS] for a description of interim and final HTTP responses.

Pushed responses are sent on a server-initiated unidirectional QUIC stream; see Section 6.2.2. A server sends zero or more interim HTTP responses, followed by a single final HTTP response, in the same manner as a standard response. Push is described in more detail in Section 4.4.

On a given stream, receipt of multiple requests or receipt of an additional HTTP response following a final HTTP response **MUST** be treated as malformed (Section 4.1.3).

An HTTP message (request or response) consists of:

1. the header section, sent as a single HEADERS frame (see Section 7.2.2),
2. optionally, the content, if present, sent as a series of DATA frames (see Section 7.2.1), and
3. optionally, the trailer section, if present, sent as a single HEADERS frame.

Header and trailer sections are described in Sections 6.3 and 6.5 of [SEMANTICS]; the content is described in Section 6.4 of [SEMANTICS].

Receipt of an invalid sequence of frames **MUST** be treated as a connection error of type H3_FRAME_UNEXPECTED; see Section 8. In particular, a DATA frame before any HEADERS frame, or a HEADERS or DATA frame after the trailing HEADERS frame, is considered invalid. Other frame types, especially unknown frame types, might be permitted subject to their own rules; see Section 9.

A server MAY send one or more PUSH_PROMISE frames (Section 7.2.5) before, after, or interleaved with the frames of a response message. These PUSH_PROMISE frames are not part of the response; see Section 4.4 for more details. PUSH_PROMISE frames are not permitted on push streams; a pushed response that includes PUSH_PROMISE frames MUST be treated as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

Frames of unknown types (Section 9), including reserved frames (Section 7.2.8) MAY be sent on a request or push stream before, after, or interleaved with other frames described in this section.

The HEADERS and PUSH_PROMISE frames might reference updates to the QPACK dynamic table. While these updates are not directly part of the message exchange, they must be received and processed before the message can be consumed. See Section 4.1.1 for more details.

Transfer codings (see Section 6.1 of [HTTP11]) are not defined for HTTP/3; the Transfer-Encoding header field MUST NOT be used.

A response MAY consist of multiple messages when and only when one or more interim responses (1xx; see Section 15.2 of [SEMANTICS]) precede a final response to the same request. Interim responses do not contain content or trailer sections.

An HTTP request/response exchange fully consumes a client-initiated bidirectional QUIC stream. After sending a request, a client MUST close the stream for sending. Unless using the CONNECT method (see Section 4.2), clients MUST NOT make stream closure dependent on receiving a response to their request. After sending a final response, the server MUST close the stream for sending. At this point, the QUIC stream is fully closed.

When a stream is closed, this indicates the end of the final HTTP message. Because some messages are large or unbounded, endpoints SHOULD begin processing partial HTTP messages once enough of the message has been received to make progress. If a client-initiated stream terminates without enough of the HTTP message to provide a complete response, the server SHOULD abort its response stream with the error code H3_REQUEST_INCOMPLETE; see Section 8.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When the server does not need to receive the remainder of the request, it MAY abort reading the request stream, send a complete response, and cleanly close the sending part of the stream. The error code H3_NO_ERROR SHOULD be used when requesting that the client stop sending on the

request stream. Clients MUST NOT discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. If the server sends a partial or complete response but does not abort reading the request, clients SHOULD continue sending the body of the request and close the stream normally.

4.1.1. Field Formatting and Compression

HTTP messages carry metadata as a series of key-value pairs called HTTP fields; see Sections 6.3 and 6.5 of [SEMANTICS]. For a listing of registered HTTP fields, see the "Hypertext Transfer Protocol (HTTP) Field Name Registry" maintained at <https://www.iana.org/assignments/http-fields/>.

Note:* This registry will not exist until [SEMANTICS] is approved. **RFC Editor, please remove this note prior to publication.

Field names are strings containing a subset of ASCII characters. Properties of HTTP field names and values are discussed in more detail in Section 5.1 of [SEMANTICS]. As in HTTP/2, characters in field names MUST be converted to lowercase prior to their encoding. A request or response containing uppercase characters in field names MUST be treated as malformed (Section 4.1.3).

Like HTTP/2, HTTP/3 does not use the Connection header field to indicate connection-specific fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/3 field section containing connection-specific fields; any message containing connection-specific fields MUST be treated as malformed (Section 4.1.3).

The only exception to this is the TE header field, which MAY be present in an HTTP/3 request header; when it is, it MUST NOT contain any value other than "trailers".

An intermediary transforming an HTTP/1.x message to HTTP/3 MUST remove connection-specific header fields as discussed in Section 7.6.1 of [SEMANTICS], or their messages will be treated by other HTTP/3 endpoints as malformed (Section 4.1.3).

4.1.1.1. Pseudo-Header Fields

Like HTTP/2, HTTP/3 employs a series of pseudo-header fields where the field name begins with the ':' character (ASCII 0x3a). These pseudo-header fields convey the target URI, the method of the request, and the status code for the response.

Pseudo-header fields are not HTTP fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document; however, an extension could negotiate a modification of this restriction; see Section 9.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailer sections. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 4.1.3).

All pseudo-header fields MUST appear in the header section before regular header fields. Any request or response that contains a pseudo-header field that appears in a header section after a regular header field MUST be treated as malformed (Section 4.1.3).

The following pseudo-header fields are defined for requests:

":method": Contains the HTTP method (Section 9 of [SEMANTICS])

":scheme": Contains the scheme portion of the target URI (Section 3.1 of [URI])

":scheme" is not restricted to URIs with scheme "http" and "https". A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

See Section 3.1.2 for guidance on using a scheme other than "https".

":authority": Contains the authority portion of the target URI (Section 3.2 of [URI]). The authority MUST NOT include the deprecated "userinfo" subcomponent for URIs of scheme "http" or "https".

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form; see Section 7.1 of [SEMANTICS]. Clients that generate HTTP/3 requests directly SHOULD use the **":authority"** pseudo-header field instead of the Host field. An intermediary that converts an HTTP/3 request to HTTP/1.1 MUST create a Host field if one is not present in a request by copying the value of the **":authority"** pseudo-header field.

`:path`: Contains the path and query parts of the target URI (the `"path-absolute"` production and optionally a `'?'` character followed by the `"query"` production; see Sections 3.3 and 3.4 of [URI]. A request in asterisk form includes the value `'*'` for the `:path` pseudo-header field.

This pseudo-header field MUST NOT be empty for `"http"` or `"https"` URIs; `"http"` or `"https"` URIs that do not contain a path component MUST include a value of `'/'`. The exception to this rule is an OPTIONS request for an `"http"` or `"https"` URI that does not include a path component; these MUST include a `:path` pseudo-header field with a value of `'*'`; see Section 7.1 of [SEMANTICS].

All HTTP/3 requests MUST include exactly one value for the `:method`, `:scheme`, and `:path` pseudo-header fields, unless it is a CONNECT request; see Section 4.2.

If the `:scheme` pseudo-header field identifies a scheme that has a mandatory authority component (including `"http"` and `"https"`), the request MUST contain either an `:authority` pseudo-header field or a `"Host"` header field. If these fields are present, they MUST NOT be empty. If both fields are present, they MUST contain the same value. If the scheme does not have a mandatory authority component and none is provided in the request target, the request MUST NOT contain the `:authority` pseudo-header or `"Host"` header fields.

An HTTP request that omits mandatory pseudo-header fields or contains invalid values for those pseudo-header fields is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

For responses, a single `:status` pseudo-header field is defined that carries the HTTP status code; see Section 15 of [SEMANTICS]. This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

4.1.1.2. Field Compression

[QPACK] describes a variation of HPACK that gives an encoder some control over how much head-of-line blocking can be caused by compression. This allows an encoder to balance compression efficiency with latency. HTTP/3 uses QPACK to compress header and trailer sections, including the pseudo-header fields present in the header section.

To allow for better compression efficiency, the "Cookie" field ([RFC6265]) MAY be split into separate field lines, each with one or more cookie-pairs, before compression. If a decompressed field section contains multiple cookie field lines, these MUST be concatenated into a single byte string using the two-byte delimiter of 0x3b, 0x20 (the ASCII string "; ") before being passed into a context other than HTTP/2 or HTTP/3, such as an HTTP/1.1 connection, or a generic HTTP server application.

4.1.1.3. Header Size Constraints

An HTTP/3 implementation MAY impose a limit on the maximum size of the message header it will accept on an individual HTTP message. A server that receives a larger header section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process. The size of a field list is calculated based on the uncompressed size of fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each field.

If an implementation wishes to advise its peer of this limit, it can be conveyed as a number of bytes in the SETTINGS_MAX_FIELD_SECTION_SIZE parameter. An implementation that has received this parameter SHOULD NOT send an HTTP message header that exceeds the indicated size, as the peer will likely refuse to process it. However, an HTTP message can traverse one or more intermediaries before reaching the origin server; see Section 3.7 of [SEMANTICS]. Because this limit is applied separately by each implementation which processes the message, messages below this limit are not guaranteed to be accepted.

4.1.2. Request Cancellation and Rejection

Once a request stream has been opened, the request MAY be cancelled by either endpoint. Clients cancel requests if the response is no longer of interest; servers cancel requests if they are unable to or choose not to respond. When possible, it is RECOMMENDED that servers send an HTTP response with an appropriate status code rather than canceling a request it has already begun processing.

Implementations SHOULD cancel requests by abruptly terminating any directions of a stream that are still open. This means resetting the sending parts of streams and aborting reading on receiving parts of streams; see Section 2.4 of [QUIC-TRANSPORT].

When the server cancels a request without performing any application processing, the request is considered "rejected." The server SHOULD abort its response stream with the error code `H3_REQUEST_REJECTED`. In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result. The client can treat requests rejected by the server as though they had never been sent at all, thereby allowing them to be retried later.

Servers MUST NOT use the `H3_REQUEST_REJECTED` error code for requests that were partially or fully processed. When a server abandons a response after partial processing, it SHOULD abort its response stream with the error code `H3_REQUEST_CANCELLED`.

Client SHOULD use the error code `H3_REQUEST_CANCELLED` to cancel requests. Upon receipt of this error code, a server MAY abruptly terminate the response using the error code `H3_REQUEST_REJECTED` if no processing was performed. Clients MUST NOT use the `H3_REQUEST_REJECTED` error code, except when a server has requested closure of the request stream with this error code.

If a stream is canceled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Only idempotent actions such as GET, PUT, or DELETE can be safely retried; a client SHOULD NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are idempotent independent of the method or some means to detect that the original request was never applied. See Section 9.2.2 of [SEMANTICS] for more details.

4.1.3. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of frames but is invalid due to:

- * the presence of prohibited fields or pseudo-header fields,
- * the absence of mandatory pseudo-header fields,
- * invalid values for pseudo-header fields,

- * pseudo-header fields after fields,
- * an invalid sequence of HTTP messages,
- * the inclusion of uppercase field names, or
- * the inclusion of invalid characters in field names or values.

A request or response that is defined as having content when it contains a Content-Length header field (Section 6.4.1 of [SEMANTICS]), is malformed if the value of a Content-Length header field does not equal the sum of the DATA frame lengths received. A response that is defined as never having content, even when a Content-Length is present, can have a non-zero Content-Length field even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 8) of type H3_MESSAGE_ERROR.

For malformed requests, a server MAY send an HTTP response indicating the error prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

4.2. The CONNECT Method

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target; see Section 9.3.6 of [SEMANTICS]. It is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2 and HTTP/3, the CONNECT method is used to establish a tunnel over a single stream.

A CONNECT request MUST be constructed as follows:

- * The ":method" pseudo-header field is set to "CONNECT"
- * The ":scheme" and ":path" pseudo-header fields are omitted

- * The ":authority" pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests; see Section 7.1 of [SEMANTICS])

The request stream remains open at the end of the request to carry the data to be transferred. A CONNECT request that does not conform to these restrictions is malformed; see Section 4.1.3.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in Section 15.3 of [SEMANTICS].

All DATA frames on the stream correspond to data sent or received on the TCP connection. The payload of any DATA frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

Once the CONNECT method has completed, only DATA frames are permitted to be sent on the stream. Extension frames MAY be used if specifically permitted by the definition of the extension. Receipt of any other known frame type MUST be treated as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will close the send stream that it sends to the client. TCP connections that remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT close a stream for sending while they still expect to receive data from the target of the CONNECT.

A TCP connection error is signaled by abruptly terminating the stream. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type H3_CONNECT_ERROR; see Section 8. Correspondingly, if a proxy detects an error with the stream or the QUIC connection, it MUST close the TCP connection. If the underlying TCP implementation permits it, the proxy SHOULD send a TCP segment with the RST bit set.

Since CONNECT creates a tunnel to an arbitrary server, proxies that support CONNECT SHOULD restrict its use to a set of known ports or a list of safe request targets; see Section 9.3.6 of [SEMANTICS] for more detail.

4.3. HTTP Upgrade

HTTP/3 does not support the HTTP Upgrade mechanism (Section 7.8 of [SEMANTICS]) or 101 (Switching Protocols) informational status code (Section 15.2.2 of [SEMANTICS]).

4.4. Server Push

Server push is an interaction mode that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. HTTP/3 server push is similar to what is described in Section 8.2 of [HTTP2], but uses different mechanisms.

Each server push is assigned a unique Push ID by the server. The Push ID is used to refer to the push in various contexts throughout the lifetime of the HTTP/3 connection.

The Push ID space begins at zero, and ends at a maximum value set by the MAX_PUSH_ID frame; see Section 7.2.7. In particular, a server is not able to push until after the client sends a MAX_PUSH_ID frame. A client sends MAX_PUSH_ID frames to control the number of pushes that a server can promise. A server SHOULD use Push IDs sequentially, beginning from zero. A client MUST treat receipt of a push stream as a connection error of type H3_ID_ERROR (Section 8) when no MAX_PUSH_ID frame has been sent or when the stream references a Push ID that is greater than the maximum Push ID.

The Push ID is used in one or more PUSH_PROMISE frames (Section 7.2.5) that carry the header section of the request message. These frames are sent on the request stream that generated the push. This allows the server push to be associated with a client request. When the same Push ID is promised on multiple request streams, the decompressed request field sections MUST contain the same fields in the same order, and both the name and the value in each field MUST be identical.

The Push ID is then included with the push stream that ultimately fulfills those promises; see Section 6.2.2. The push stream identifies the Push ID of the promise that it fulfills, then contains a response to the promised request as described in Section 4.1.

Finally, the Push ID can be used in CANCEL_PUSH frames; see Section 7.2.3. Clients use this frame to indicate they do not wish to receive a promised resource. Servers use this frame to indicate they will not be fulfilling a previous promise.

Not all requests can be pushed. A server MAY push requests that have the following properties:

- * cacheable; see Section 9.2.3 of [SEMANTICS]
- * safe; see Section 9.2.1 of [SEMANTICS]
- * does not include a request body or trailer section

The server MUST include a value in the ":authority" pseudo-header field for which the server is authoritative. If the client has not yet validated the connection for the origin indicated by the pushed request, it MUST perform the same verification process it would do before sending a request for that origin on the connection; see Section 3.3. If this verification fails, the client MUST NOT consider the server authoritative for that origin.

Clients SHOULD send a CANCEL_PUSH frame upon receipt of a PUSH_PROMISE frame carrying a request that is not cacheable, is not known to be safe, that indicates the presence of a request body, or for which it does not consider the server authoritative. Any corresponding responses MUST NOT be used or cached.

Each pushed response is associated with one or more client requests. The push is associated with the request stream on which the PUSH_PROMISE frame was received. The same server push can be associated with additional client requests using a PUSH_PROMISE frame with the same Push ID on multiple request streams. These associations do not affect the operation of the protocol, but MAY be considered by user agents when deciding how to use pushed resources.

Ordering of a PUSH_PROMISE frame in relation to certain parts of the response is important. The server SHOULD send PUSH_PROMISE frames prior to sending HEADERS or DATA frames that reference the promised responses. This reduces the chance that a client requests a resource that will be pushed by the server.

Due to reordering, push stream data can arrive before the corresponding PUSH_PROMISE frame. When a client receives a new push stream with an as-yet-unknown Push ID, both the associated client request and the pushed request header fields are unknown. The client can buffer the stream data in expectation of the matching PUSH_PROMISE. The client can use stream flow control (see Section 4.1 of [QUIC-TRANSPORT]) to limit the amount of data a server may commit to the pushed stream.

Push stream data can also arrive after a client has canceled a push. In this case, the client can abort reading the stream with an error code of H3_REQUEST_CANCELLED. This asks the server not to transfer additional data and indicates that it will be discarded upon receipt.

Pushed responses that are cacheable (see Section 3 of [CACHING]) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present; see Section 5.2.2.3 of [CACHING]) at the time the pushed response is received.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

5. Connection Closure

Once established, an HTTP/3 connection can be used for many requests and responses over time until the connection is closed. Connection closure can happen in any of several different ways.

5.1. Idle Connections

Each QUIC endpoint declares an idle timeout during the handshake. If the QUIC connection remains idle (no packets received) for longer than this duration, the peer will assume that the connection has been closed. HTTP/3 implementations will need to open a new HTTP/3 connection for new requests if the existing connection has been idle for longer than the idle timeout negotiated during the QUIC handshake, and SHOULD do so if approaching the idle timeout; see Section 10.1 of [QUIC-TRANSPORT].

HTTP clients are expected to request that the transport keep connections open while there are responses outstanding for requests or server pushes, as described in Section 10.1.2 of [QUIC-TRANSPORT]. If the client is not expecting a response from the server, allowing an idle connection to time out is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY

maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers. Servers SHOULD NOT actively keep connections open.

5.2. Connection Shutdown

Even when a connection is not idle, either endpoint can decide to stop using the connection and initiate a graceful connection close. Endpoints initiate the graceful shutdown of an HTTP/3 connection by sending a GOAWAY frame (Section 7.2.6). The GOAWAY frame contains an identifier that indicates to the receiver the range of requests or pushes that were or might be processed in this connection. The server sends a client-initiated bidirectional Stream ID; the client sends a Push ID (Section 4.4). Requests or pushes with the indicated identifier or greater are rejected (Section 4.1.2) by the sender of the GOAWAY. This identifier MAY be zero if no requests or pushes were processed.

The information in the GOAWAY frame enables a client and server to agree on which requests or pushes were accepted prior to the shutdown of the HTTP/3 connection. Upon sending a GOAWAY frame, the endpoint SHOULD explicitly cancel (see Section 4.1.2 and Section 7.2.3) any requests or pushes that have identifiers greater than or equal to that indicated, in order to clean up transport state for the affected streams. The endpoint SHOULD continue to do so as more requests or pushes arrive.

Endpoints MUST NOT initiate new requests or promise new pushes on the connection after receipt of a GOAWAY frame from the peer. Clients MAY establish a new connection to send additional requests.

Some requests or pushes might already be in transit:

- * Upon receipt of a GOAWAY frame, if the client has already sent requests with a Stream ID greater than or equal to the identifier contained in the GOAWAY frame, those requests will not be processed. Clients can safely retry unprocessed requests on a different HTTP connection. A client that is unable to retry requests loses all requests that are in flight when the server closes the connection.

Requests on Stream IDs less than the Stream ID in a GOAWAY frame from the server might have been processed; their status cannot be known until a response is received, the stream is reset individually, another GOAWAY is received with a lower Stream ID than that of the request in question, or the connection terminates.

Servers MAY reject individual requests on streams below the indicated ID if these requests were not processed.

- * If a server receives a GOAWAY frame after having promised pushes with a Push ID greater than or equal to the identifier contained in the GOAWAY frame, those pushes will not be accepted.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a request has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint MAY send multiple GOAWAY frames indicating different identifiers, but the identifier in each frame MUST NOT be greater than the identifier in any previous frame, since clients might already have retried unprocessed requests on another HTTP connection. Receiving a GOAWAY containing a larger identifier than previously received MUST be treated as a connection error of type H3_ID_ERROR; see Section 8.

An endpoint that is attempting to gracefully shut down a connection can send a GOAWAY frame with a value set to the maximum possible value ($2^{62}-4$ for servers, $2^{62}-1$ for clients). This ensures that the peer stops creating new requests or pushes. After allowing time for any in-flight requests or pushes to arrive, the endpoint can send another GOAWAY frame indicating which requests or pushes it might accept before the end of the connection. This ensures that a connection can be cleanly shut down without losing requests.

A client has more flexibility in the value it chooses for the Push ID in a GOAWAY that it sends. A value of $2^{62}-1$ indicates that the server can continue fulfilling pushes that have already been promised. A smaller value indicates the client will reject pushes with Push IDs greater than or equal to this value. Like the server, the client MAY send subsequent GOAWAY frames so long as the specified Push ID is no greater than any previously sent value.

Even when a GOAWAY indicates that a given request or push will not be processed or accepted upon receipt, the underlying transport resources still exist. The endpoint that initiated these requests can cancel them to clean up transport state.

Once all accepted requests and pushes have been processed, the endpoint can permit the connection to become idle, or MAY initiate an immediate closure of the connection. An endpoint that completes a graceful shutdown SHOULD use the H3_NO_ERROR error code when closing the connection.

If a client has consumed all available bidirectional stream IDs with requests, the server need not send a GOAWAY frame, since the client is unable to make further requests.

5.3. Immediate Application Closure

An HTTP/3 implementation can immediately close the QUIC connection at any time. This results in sending a QUIC CONNECTION_CLOSE frame to the peer indicating that the application layer has terminated the connection. The application error code in this frame indicates to the peer why the connection is being closed. See Section 8 for error codes that can be used when closing a connection in HTTP/3.

Before closing the connection, a GOAWAY frame MAY be sent to allow the client to retry some requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION_CLOSE frame improves the chances of the frame being received by clients.

If there are open streams that have not been explicitly closed, they are implicitly closed when the connection is closed; see Section 10.2 of [QUIC-TRANSPORT].

5.4. Transport Closure

For various reasons, the QUIC transport could indicate to the application layer that the connection has terminated. This might be due to an explicit closure by the peer, a transport-level error, or a change in network topology that interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST assume that any request that was sent, whether in whole or in part, might have been processed.

6. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. In version 1 of QUIC, the stream data containing HTTP frames is carried by QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received stream data, exposing a reliable byte stream to the application. Although QUIC permits out-of-order delivery within a stream, HTTP/3 does not make use of this feature.

QUIC streams can be either unidirectional, carrying data only from initiator to receiver, or bidirectional. Streams can be initiated by either the client or the server. For more detail on QUIC streams, see Section 2 of [QUIC-TRANSPORT].

When HTTP fields and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction or to the entire HTTP/3 connection context.

6.1. Bidirectional Streams

All client-initiated bidirectional streams are used for HTTP requests and responses. A bidirectional stream ensures that the response can be readily correlated with the request. These streams are referred to as request streams.

This means that the client's first request occurs on QUIC stream 0, with subsequent requests on stream 4, 8, and so on. In order to permit these streams to open, an HTTP/3 server SHOULD configure non-zero minimum values for the number of permitted streams and the initial stream flow control window. So as to not unnecessarily limit parallelism, at least 100 request streams SHOULD be permitted at a time.

HTTP/3 does not use server-initiated bidirectional streams, though an extension could define a use for these streams. Clients MUST treat receipt of a server-initiated bidirectional stream as a connection error of type H3_STREAM_CREATION_ERROR (Section 8) unless such an extension has been negotiated.

6.2. Unidirectional Streams

Unidirectional streams, in either direction, are used for a range of purposes. The purpose is indicated by a stream type, which is sent as a variable-length integer at the start of the stream. The format and structure of data that follows this integer is determined by the stream type.

```
Unidirectional Stream Header {  
    Stream Type (i),  
}
```

Figure 1: Unidirectional Stream Header

Two stream types are defined in this document: control streams (Section 6.2.1) and push streams (Section 6.2.2). [QPACK] defines two additional stream types. Other stream types can be defined by extensions to HTTP/3; see Section 9 for more details. Some stream types are reserved (Section 6.2.3).

The performance of HTTP/3 connections in the early phase of their lifetime is sensitive to the creation and exchange of data on unidirectional streams. Endpoints that excessively restrict the number of streams or the flow control window of these streams will increase the chance that the remote peer reaches the limit early and becomes blocked. In particular, implementations should consider that remote peers may wish to exercise reserved stream behavior (Section 6.2.3) with some of the unidirectional streams they are permitted to use. To avoid blocking, the transport parameters sent by both clients and servers **MUST** allow the peer to create at least one unidirectional stream for the HTTP control stream plus the number of unidirectional streams required by mandatory extensions (three being the minimum number required for the base HTTP/3 protocol and QPACK), and **SHOULD** provide at least 1,024 bytes of flow control credit to each stream.

Note that an endpoint is not required to grant additional credits to create more unidirectional streams if its peer consumes all the initial credits before creating the critical unidirectional streams. Endpoints **SHOULD** create the HTTP control stream as well as the unidirectional streams required by mandatory extensions (such as the QPACK encoder and decoder streams) first, and then create additional streams as allowed by their peer.

If the stream header indicates a stream type that is not supported by the recipient, the remainder of the stream cannot be consumed as the semantics are unknown. Recipients of unknown stream types MAY abort reading of the stream with an error code of `H3_STREAM_CREATION_ERROR` or a reserved error code (Section 8.1), but MUST NOT consider such streams to be a connection error of any kind.

Implementations MAY send stream types before knowing whether the peer supports them. However, stream types that could modify the state or semantics of existing protocol components, including QPACK or other extensions, MUST NOT be sent until the peer is known to support them.

A sender can close or reset a unidirectional stream unless otherwise specified. A receiver MUST tolerate unidirectional streams being closed or reset prior to the reception of the unidirectional stream header.

6.2.1. Control Streams

A control stream is indicated by a stream type of `0x00`. Data on this stream consists of HTTP/3 frames, as defined in Section 7.2.

Each side MUST initiate a single control stream at the beginning of the connection and send its SETTINGS frame as the first frame on this stream. If the first frame of the control stream is any other frame type, this MUST be treated as a connection error of type `H3_MISSING_SETTINGS`. Only one control stream per peer is permitted; receipt of a second stream claiming to be a control stream MUST be treated as a connection error of type `H3_STREAM_CREATION_ERROR`. The sender MUST NOT close the control stream, and the receiver MUST NOT request that the sender close the control stream. If either control stream is closed at any point, this MUST be treated as a connection error of type `H3_CLOSED_CRITICAL_STREAM`. Connection errors are described in Section 8.

Because the contents of the control stream are used to manage the behavior of other streams, endpoints SHOULD provide enough flow control credit to keep the peer's control stream from becoming blocked.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon as it is able. Depending on whether 0-RTT is available on the QUIC connection, either client or server might be able to send stream data first.

6.2.2. Push Streams

Server push is an optional feature introduced in HTTP/2 that allows a server to initiate a response before a request has been made. See Section 4.4 for more details.

A push stream is indicated by a stream type of 0x01, followed by the Push ID of the promise that it fulfills, encoded as a variable-length integer. The remaining data on this stream consists of HTTP/3 frames, as defined in Section 7.2, and fulfills a promised server push by zero or more interim HTTP responses followed by a single final HTTP response, as defined in Section 4.1. Server push and Push IDs are described in Section 4.4.

Only servers can push; if a server receives a client-initiated push stream, this MUST be treated as a connection error of type H3_STREAM_CREATION_ERROR; see Section 8.

```
Push Stream Header {  
    Stream Type (i) = 0x01,  
    Push ID (i),  
}
```

Figure 2: Push Stream Header

Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type H3_ID_ERROR; see Section 8.

6.2.3. Reserved Stream Types

Stream types of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored. These streams have no semantics, and can be sent when application-layer padding is desired. They MAY also be sent on connections where no data is currently being transferred. Endpoints MUST NOT consider these streams to have any meaning upon receipt.

The payload and length of the stream are selected in any manner the sending implementation chooses. When sending a reserved stream type, the implementation MAY either terminate the stream cleanly or reset it. When resetting the stream, either the H3_NO_ERROR error code or a reserved error code (Section 8.1) SHOULD be used.

7. HTTP Framing Layer

HTTP frames are carried on QUIC streams, as described in Section 6. HTTP/3 defines three stream types: control stream, request stream, and push stream. This section describes HTTP/3 frame formats and their permitted stream types; see Table 1 for an overview. A comparison between HTTP/2 and HTTP/3 frames is provided in Appendix A.2.

Frame	Control Stream	Request Stream	Push Stream	Section
DATA	No	Yes	Yes	Section 7.2.1
HEADERS	No	Yes	Yes	Section 7.2.2
CANCEL_PUSH	Yes	No	No	Section 7.2.3
SETTINGS	Yes (1)	No	No	Section 7.2.4
PUSH_PROMISE	No	Yes	No	Section 7.2.5
GOAWAY	Yes	No	No	Section 7.2.6
MAX_PUSH_ID	Yes	No	No	Section 7.2.7
Reserved	Yes	Yes	Yes	Section 7.2.8

Table 1: HTTP/3 Frames and Stream Type Overview

The SETTINGS frame can only occur as the first frame of a Control stream; this is indicated in Table 1 with a (1). Specific guidance is provided in the relevant section.

Note that, unlike QUIC frames, HTTP/3 frames can span multiple packets.

7.1. Frame Layout

All frames have the following format:

```
HTTP/3 Frame Format {  
    Type (i),  
    Length (i),  
    Frame Payload (...),  
}
```

Figure 3: HTTP/3 Frame Format

A frame includes the following fields:

Type: A variable-length integer that identifies the frame type.

Length: A variable-length integer that describes the length in bytes of the Frame Payload.

Frame Payload: A payload, the semantics of which are determined by the Type field.

Each frame's payload MUST contain exactly the fields identified in its description. A frame payload that contains additional bytes after the identified fields or a frame payload that terminates before the end of the identified fields MUST be treated as a connection error of type `H3_FRAME_ERROR`; see Section 8. In particular, redundant length encodings MUST be verified to be self-consistent; see Section 10.8.

When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error of type `H3_FRAME_ERROR`; see Section 8. Streams that terminate abruptly may be reset at any point in a frame.

7.2. Frame Definitions

7.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of bytes associated with HTTP request or response content.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on a control stream, the recipient MUST respond with a connection error of type `H3_FRAME_UNEXPECTED`; see Section 8.

```
DATA Frame {  
    Type (i) = 0x0,  
    Length (i),  
    Data (...),  
}
```

Figure 4: DATA Frame

7.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry an HTTP field section, encoded using QPACK. See [QPACK] for more details.

```
HEADERS Frame {  
    Type (i) = 0x1,  
    Length (i),  
    Encoded Field Section (...),  
}
```

Figure 5: HEADERS Frame

HEADERS frames can only be sent on request or push streams. If a HEADERS frame is received on a control stream, the recipient MUST respond with a connection error (Section 8) of type H3_FRAME_UNEXPECTED.

7.2.3. CANCEL_PUSH

The CANCEL_PUSH frame (type=0x3) is used to request cancellation of a server push prior to the push stream being received. The CANCEL_PUSH frame identifies a server push by Push ID (see Section 4.4), encoded as a variable-length integer.

When a client sends CANCEL_PUSH, it is indicating that it does not wish to receive the promised resource. The server SHOULD abort sending the resource, but the mechanism to do so depends on the state of the corresponding push stream. If the server has not yet created a push stream, it does not create one. If the push stream is open, the server SHOULD abruptly terminate that stream. If the push stream has already ended, the server MAY still abruptly terminate the stream or MAY take no action.

A server sends CANCEL_PUSH to indicate that it will not be fulfilling a promise which was previously sent. The client cannot expect the corresponding promise to be fulfilled, unless it has already received and processed the promised response. Regardless of whether a push stream has been opened, a server SHOULD send a CANCEL_PUSH frame when it determines that promise will not be fulfilled. If a stream has already been opened, the server can abort sending on the stream with an error code of H3_REQUEST_CANCELLED.

Sending a CANCEL_PUSH frame has no direct effect on the state of existing push streams. A client SHOULD NOT send a CANCEL_PUSH frame when it has already received a corresponding push stream. A push stream could arrive after a client has sent a CANCEL_PUSH frame, because a server might not have processed the CANCEL_PUSH. The client SHOULD abort reading the stream with an error code of H3_REQUEST_CANCELLED.

A CANCEL_PUSH frame is sent on the control stream. Receiving a CANCEL_PUSH frame on a stream other than the control stream MUST be treated as a connection error of type H3_FRAME_UNEXPECTED.

```
CANCEL_PUSH Frame {  
    Type (i) = 0x3,  
    Length (i),  
    Push ID (i),  
}
```

Figure 6: CANCEL_PUSH Frame

The CANCEL_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled; see Section 4.4. If a CANCEL_PUSH frame is received that references a Push ID greater than currently allowed on the connection, this MUST be treated as a connection error of type H3_ID_ERROR.

If the client receives a CANCEL_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH_PROMISE frame due to reordering. If a server receives a CANCEL_PUSH frame for a Push ID that has not yet been mentioned by a PUSH_PROMISE frame, this MUST be treated as a connection error of type H3_ID_ERROR.

7.2.4. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. Individually, a SETTINGS parameter can also be referred to as a "setting"; the identifier and value of each setting parameter can be referred to as a "setting identifier" and a "setting value".

SETTINGS frames always apply to an entire HTTP/3 connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of each control stream (see Section 6.2.1) by each peer, and MUST NOT be sent subsequently. If an endpoint receives a second SETTINGS frame on the control stream, the endpoint MUST respond with a connection error of type H3_FRAME_UNEXPECTED.

SETTINGS frames MUST NOT be sent on any stream other than the control stream. If an endpoint receives a SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type H3_FRAME_UNEXPECTED.

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer that can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - each peer uses SETTINGS to advertise a set of supported values. The definition of the setting would describe how each peer combines the two sets to conclude which choice will be used. SETTINGS does not provide a mechanism to identify when the choice takes effect.

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume a very large response field section, while servers are more cautious about request size.

The same setting identifier MUST NOT occur more than once in the SETTINGS frame. A receiver MAY treat the presence of duplicate setting identifiers as a connection error of type H3_SETTINGS_ERROR.

The payload of a SETTINGS frame consists of zero or more parameters. Each parameter consists of a setting identifier and a value, both encoded as QUIC variable-length integers.

```
Setting {  
    Identifier (i),  
    Value (i),  
}  
  
SETTINGS Frame {  
    Type (i) = 0x4,  
    Length (i),  
    Setting (...) ...,  
}
```

Figure 7: SETTINGS Frame

An implementation MUST ignore any parameter with an identifier it does not understand.

7.2.4.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/3:

SETTINGS_MAX_FIELD_SECTION_SIZE (0x6): The default value is unlimited. See Section 4.1.1.3 for usage.

Setting identifiers of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown identifiers be ignored. Such settings have no defined meaning. Endpoints SHOULD include at least one such setting in their SETTINGS frame. Endpoints MUST NOT consider such settings to have any meaning upon receipt.

Because the setting has no defined meaning, the value of the setting can be any value the implementation selects.

Setting identifiers which were defined in [HTTP2] where there is no corresponding HTTP/3 setting have also been reserved (Section 11.2.2). These reserved settings MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3_SETTINGS_ERROR.

Additional settings can be defined by extensions to HTTP/3; see Section 9 for more details.

7.2.4.2. Initialization

An HTTP implementation MUST NOT send frames or requests that would be invalid based on its current understanding of the peer's settings.

All settings begin at an initial value. Each endpoint SHOULD use these initial values to send messages before the peer's SETTINGS frame has arrived, as packets carrying the settings can be lost or delayed. When the SETTINGS frame arrives, any settings are changed to their new values.

This removes the need to wait for the SETTINGS frame before sending messages. Endpoints MUST NOT require any data to be received from the peer prior to sending the SETTINGS frame; settings MUST be sent as soon as the transport is ready to send data.

For servers, the initial value of each client setting is the default value.

For clients using a 1-RTT QUIC connection, the initial value of each server setting is the default value. 1-RTT keys will always become available prior to the packet containing SETTINGS being processed by QUIC, even if the server sends SETTINGS immediately. Clients SHOULD NOT wait indefinitely for SETTINGS to arrive before sending requests, but SHOULD process received datagrams in order to increase the likelihood of processing SETTINGS before sending the first request.

When a 0-RTT QUIC connection is being used, the initial value of each server setting is the value used in the previous session. Clients SHOULD store the settings the server provided in the HTTP/3 connection where resumption information was provided, but MAY opt not to store settings in certain cases (e.g., if the session ticket is received before the SETTINGS frame). A client MUST comply with stored settings -- or default values, if no values are stored -- when attempting 0-RTT. Once a server has provided new settings, clients MUST comply with those values.

A server can remember the settings that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the HTTP/3 settings values in determining whether to accept 0-RTT data. If the server cannot determine that the settings remembered by a client are compatible with its current settings, it MUST NOT accept 0-RTT data. Remembered settings are compatible if a client complying with those settings would not violate the server's current settings.

A server MAY accept 0-RTT and subsequently provide different settings in its SETTINGS frame. If 0-RTT data is accepted by the server, its SETTINGS frame MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. The server MUST include all settings that differ from their default values. If a server accepts 0-RTT but then sends settings that are not compatible with the previously specified settings, this MUST be treated as a

connection error of type `H3_SETTINGS_ERROR`. If a server accepts 0-RTT but then sends a `SETTINGS` frame that omits a setting value that the client understands (apart from reserved setting identifiers) that was previously specified to have a non-default value, this **MUST** be treated as a connection error of type `H3_SETTINGS_ERROR`.

7.2.5. `PUSH_PROMISE`

The `PUSH_PROMISE` frame (type=0x5) is used to carry a promised request header section from server to client on a request stream, as in HTTP/2.

```
PUSH_PROMISE Frame {  
    Type (i) = 0x5,  
    Length (i),  
    Push ID (i),  
    Encoded Field Section (..),  
}
```

Figure 8: `PUSH_PROMISE` Frame

The payload consists of:

Push ID: A variable-length integer that identifies the server push operation. A Push ID is used in push stream headers (Section 4.4) and `CANCEL_PUSH` frames (Section 7.2.3).

Encoded Field Section: QPACK-encoded request header fields for the promised response. See [QPACK] for more details.

A server **MUST NOT** use a Push ID that is larger than the client has provided in a `MAX_PUSH_ID` frame (Section 7.2.7). A client **MUST** treat receipt of a `PUSH_PROMISE` frame that contains a larger Push ID than the client has advertised as a connection error of `H3_ID_ERROR`.

A server **MAY** use the same Push ID in multiple `PUSH_PROMISE` frames. If so, the decompressed request header sets **MUST** contain the same fields in the same order, and both the name and the value in each field **MUST** be exact matches. Clients **SHOULD** compare the request header sections for resources promised multiple times. If a client receives a Push ID that has already been promised and detects a mismatch, it **MUST** respond with a connection error of type `H3_GENERAL_PROTOCOL_ERROR`. If the decompressed field sections match exactly, the client **SHOULD** associate the pushed content with each stream on which a `PUSH_PROMISE` frame was received.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH_PROMISE frame that uses a Push ID that they have already consumed and discarded are forced to ignore the promise.

If a PUSH_PROMISE frame is received on the control stream, the client MUST respond with a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

A client MUST NOT send a PUSH_PROMISE frame. A server MUST treat the receipt of a PUSH_PROMISE frame as a connection error of type H3_FRAME_UNEXPECTED; see Section 8.

See Section 4.4 for a description of the overall server push mechanism.

7.2.6. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of an HTTP/3 connection by either endpoint. GOAWAY allows an endpoint to stop accepting new requests or pushes while still finishing processing of previously received requests and pushes. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

```
GOAWAY Frame {  
    Type (i) = 0x7,  
    Length (i),  
    Stream ID/Push ID (...),  
}
```

Figure 9: GOAWAY Frame

The GOAWAY frame is always sent on the control stream. In the server to client direction, it carries a QUIC Stream ID for a client-initiated bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type H3_ID_ERROR.

In the client to server direction, the GOAWAY frame carries a Push ID encoded as a variable-length integer.

The GOAWAY frame applies to the entire connection, not a specific stream. A client **MUST** treat a GOAWAY frame on a stream other than the control stream as a connection error of type `H3_FRAME_UNEXPECTED`; see Section 8.

See Section 5.2 for more information on the use of the GOAWAY frame.

7.2.7. MAX_PUSH_ID

The `MAX_PUSH_ID` frame (type=0xd) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in `PUSH_PROMISE` and `CANCEL_PUSH` frames. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit maintained by the QUIC transport.

The `MAX_PUSH_ID` frame is always sent on the control stream. Receipt of a `MAX_PUSH_ID` frame on any other stream **MUST** be treated as a connection error of type `H3_FRAME_UNEXPECTED`.

A server **MUST NOT** send a `MAX_PUSH_ID` frame. A client **MUST** treat the receipt of a `MAX_PUSH_ID` frame as a connection error of type `H3_FRAME_UNEXPECTED`.

The maximum Push ID is unset when an HTTP/3 connection is created, meaning that a server cannot push until it receives a `MAX_PUSH_ID` frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending `MAX_PUSH_ID` frames as the server fulfills or cancels server pushes.

```
MAX_PUSH_ID Frame {  
    Type (i) = 0xd,  
    Length (i),  
    Push ID (i),  
}
```

Figure 10: `MAX_PUSH_ID` Frame

The `MAX_PUSH_ID` frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use; see Section 4.4. A `MAX_PUSH_ID` frame cannot reduce the maximum Push ID; receipt of a `MAX_PUSH_ID` frame that contains a smaller value than previously received **MUST** be treated as a connection error of type `H3_ID_ERROR`.

7.2.8. Reserved Frame Types

Frame types of the format `"0x1f * N + 0x21"` for non-negative integer values of `N` are reserved to exercise the requirement that unknown types be ignored (Section 9). These frames have no semantics, and MAY be sent on any stream where frames are allowed to be sent. This enables their use for application-layer padding. Endpoints MUST NOT consider these frames to have any meaning upon receipt.

The payload and length of the frames are selected in any manner the implementation chooses.

Frame types that were used in HTTP/2 where there is no corresponding HTTP/3 frame have also been reserved (Section 11.2.1). These frame types MUST NOT be sent, and their receipt MUST be treated as a connection error of type `H3_FRAME_UNEXPECTED`.

8. Error Handling

When a stream cannot be completed successfully, QUIC allows the application to abruptly terminate (reset) that stream and communicate a reason; see Section 2.4 of [QUIC-TRANSPORT]. This is referred to as a "stream error." An HTTP/3 implementation can decide to close a QUIC stream and communicate the type of error. Wire encodings of error codes are defined in Section 8.1. Stream errors are distinct from HTTP status codes which indicate error conditions. Stream errors indicate that the sender did not transfer or consume the full request or response, while HTTP status codes indicate the result of a request that was successfully received.

If an entire connection needs to be terminated, QUIC similarly provides mechanisms to communicate a reason; see Section 5.3 of [QUIC-TRANSPORT]. This is referred to as a "connection error." Similar to stream errors, an HTTP/3 implementation can terminate a QUIC connection and communicate the reason using an error code from Section 8.1.

Although the reasons for closing streams and connections are called "errors," these actions do not necessarily indicate a problem with the connection or either implementation. For example, a stream can be reset if the requested resource is no longer needed.

An endpoint MAY choose to treat a stream error as a connection error under certain circumstances, closing the entire connection in response to a condition on a single stream. Implementations need to consider the impact on outstanding requests before making this choice.

Because new error codes can be defined without negotiation (see Section 9), use of an error code in an unexpected context or receipt of an unknown error code MUST be treated as equivalent to H3_NO_ERROR. However, closing a stream can have other effects regardless of the error code; for example, see Section 4.1.

8.1. HTTP/3 Error Codes

The following error codes are defined for use when abruptly terminating streams, aborting reading of streams, or immediately closing HTTP/3 connections.

H3_NO_ERROR (0x100): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

H3_GENERAL_PROTOCOL_ERROR (0x101): Peer violated protocol requirements in a way that does not match a more specific error code, or endpoint declines to use the more specific error code.

H3_INTERNAL_ERROR (0x102): An internal error has occurred in the HTTP stack.

H3_STREAM_CREATION_ERROR (0x103): The endpoint detected that its peer created a stream that it will not accept.

H3_CLOSED_CRITICAL_STREAM (0x104): A stream required by the HTTP/3 connection was closed or reset.

H3_FRAME_UNEXPECTED (0x105): A frame was received that was not permitted in the current state or on the current stream.

H3_FRAME_ERROR (0x106): A frame that fails to satisfy layout requirements or with an invalid size was received.

H3_EXCESSIVE_LOAD (0x107): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

H3_ID_ERROR (0x108): A Stream ID or Push ID was used incorrectly, such as exceeding a limit, reducing a limit, or being reused.

H3_SETTINGS_ERROR (0x109): An endpoint detected an error in the payload of a SETTINGS frame.

H3_MISSING_SETTINGS (0x10a): No SETTINGS frame was received at the beginning of the control stream.

H3_REQUEST_REJECTED (0x10b): A server rejected a request without performing any application processing.

H3_REQUEST_CANCELLED (0x10c): The request or its response (including pushed response) is cancelled.

H3_REQUEST_INCOMPLETE (0x10d): The client's stream terminated without containing a fully-formed request.

H3_MESSAGE_ERROR (0x10e): An HTTP message was malformed and cannot be processed.

H3_CONNECT_ERROR (0x10f): The TCP connection established in response to a CONNECT request was reset or abnormally closed.

H3_VERSION_FALLBACK (0x110): The requested operation cannot be served over HTTP/3. The peer should retry over HTTP/1.1.

Error codes of the format "0x1f * N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown error codes be treated as equivalent to H3_NO_ERROR (Section 9). Implementations SHOULD select an error code from this space with some probability when they would have sent H3_NO_ERROR.

9. Extensions to HTTP/3

HTTP/3 permits extension of the protocol. Within the limitations described in this section, protocol extensions can be used to provide additional services or alter any aspect of the protocol. Extensions are effective only within the scope of a single HTTP/3 connection.

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or fields.

Extensions are permitted to use new frame types (Section 7.2), new settings (Section 7.2.4.1), new error codes (Section 8), or new unidirectional stream types (Section 6.2). Registries are established for managing these extension points: frame types (Section 11.2.1), settings (Section 11.2.2), error codes (Section 11.2.3), and stream types (Section 11.2.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames and abort reading on unidirectional streams that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, where a known frame type is required to be in a specific location, such as the SETTINGS frame as the first frame of the control stream (see Section 6.2.1), an unknown frame type does not satisfy that requirement and SHOULD be treated as an error.

Extensions that could change the semantics of existing protocol components **MUST** be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. Coordinating when such a revised layout comes into effect could prove complex. As such, allocating new identifiers for new definitions of existing protocol elements is likely to be more effective.

This document does not mandate a specific method for negotiating the use of an extension but notes that a setting (Section 7.2.4.1) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the default value **MUST** be defined in such a fashion that the extension is disabled if the setting is omitted.

10. Security Considerations

The security considerations of HTTP/3 should be comparable to those of HTTP/2 with TLS. However, many of the considerations from Section 10 of [HTTP2] apply to [QUIC-TRANSPORT] and are discussed in that document.

10.1. Server Authority

HTTP/3 relies on the HTTP definition of authority. The security considerations of establishing authority are discussed in Section 17.1 of [SEMANTICS].

10.2. Cross-Protocol Attacks

The use of ALPN in the TLS and QUIC handshakes establishes the target application protocol before application-layer bytes are processed. This ensures that endpoints have strong assurances that peers are using the same protocol.

This does not guarantee protection from all cross-protocol attacks. Section 21.5 of [QUIC-TRANSPORT] describes some ways in which the plaintext of QUIC packets can be used to perform request forgery against endpoints that don't use authenticated transports.

10.3. Intermediary Encapsulation Attacks

The HTTP/3 field encoding allows the expression of names that are not valid field names in the syntax used by HTTP (Section 5.1 of [SEMANTICS]). Requests or responses containing invalid field names MUST be treated as malformed (Section 4.1.3). An intermediary therefore cannot translate an HTTP/3 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/3 can transport field values that are not valid. While most values that can be encoded will not alter field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a field value MUST be treated as malformed (Section 4.1.3). Valid characters are defined by the "field-content" ABNF rule in Section 5.5 of [SEMANTICS].

10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server MUST ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Clients are required to reject pushed responses for which an origin server is not authoritative; see Section 4.4.

10.5. Denial-of-Service Considerations

An HTTP/3 connection can demand a greater commitment of resources to operate than an HTTP/1.1 or HTTP/2 connection. The use of field compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH_PROMISE frames is constrained in a similar fashion. A client that accepts server push SHOULD limit the number of Push IDs it issues at a time.

Processing capacity cannot be guarded as effectively as state capacity.

The ability to send undefined protocol elements that the peer is required to ignore can be abused to cause a peer to expend additional processing time. This might be done by setting multiple undefined SETTINGS parameters, unknown frame types, or unknown stream types. Note, however, that some uses are entirely legitimate, such as optional-to-understand extensions and padding to increase resistance to traffic analysis.

Compression of field sections also offers some opportunities to waste processing resources; see Section 7 of [QPACK] for more details on potential abuses.

All these features -- i.e., server push, unknown protocol elements, field compression -- have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that does not monitor such behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error of type H3_EXCESSIVE_LOAD (Section 8), but false positives will result in disrupting valid connections and requests.

10.5.1. Limits on Field Section Size

A large field section (Section 4.1) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header section, which prevents streaming of the header section to its ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean that an endpoint likely needs to buffer the entire header section. Since there is no hard limit to the size of a field section, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the SETTINGS_MAX_FIELD_SECTION_SIZE (Section 4.1.1.3) setting to advise peers of limits that might apply on the size of field sections. This setting is only advisory, so endpoints MAY choose to send field sections that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to an HTTP/3 connection, so any request or

response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger field section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process.

10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on a proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. Therefore, a proxy that supports CONNECT might be more conservative in the number of simultaneous requests it accepts.

A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME_WAIT state. To account for this, a proxy might delay increasing the QUIC stream limits for some time after a TCP connection terminates.

10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/3 enables compression of fields (Section 4.1.1); the following concerns also apply to the use of HTTP compressed content-codings; see Section 8.4.1 of [SEMANTICS].

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression contexts are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined.

Further considerations regarding the compression of field sections are described in [QPACK].

10.7. Padding and Traffic Analysis

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Where HTTP/2 employs PADDING frames and Padding fields in other frames to make a connection more resistant to traffic analysis, HTTP/3 can either rely on transport-layer padding or employ the reserved frame and stream types discussed in Section 7.2.8 and Section 6.2.3. These methods of padding produce different results in terms of the granularity of padding, how padding is arranged in relation to the information that is being protected, whether padding is applied in the case of packet loss, and how an implementation might control padding.

Reserved stream types can be used to give the appearance of sending traffic even when the connection is idle. Because HTTP traffic often occurs in bursts, apparent traffic can be used to obscure the timing or duration of such bursts, even to the point of appearing to send a constant stream of data. However, as such traffic is still flow controlled by the receiver, a failure to promptly drain such streams and provide additional flow control credit can limit the sender's ability to send real traffic.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Use of padding can result in less protection than might seem immediately obvious. Redundant padding could even be counterproductive. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

10.8. Frame Parsing

Several protocol elements contain nested length elements, typically in the form of frames with an explicit length containing variable-length integers. This could pose a security risk to an incautious implementer. An implementation MUST ensure that the length of a frame exactly matches the length of the fields it contains.

10.9. Early Data

The use of 0-RTT with HTTP/3 creates an exposure to replay attack. The anti-replay mitigations in [HTTP-REPLAY] MUST be applied when using HTTP/3 with 0-RTT. When applying [HTTP-REPLAY] to HTTP/3, references to the TLS layer refer to the handshake performed within QUIC, while all references to application data refer to the contents of streams.

10.10. Migration

Certain HTTP implementations use the client address for logging or access-control purposes. Since a QUIC client's address might change during a connection (and future versions might support simultaneous use of multiple addresses), such implementations will need to either actively retrieve the client's current address or addresses when they are relevant or explicitly accept that the original address might change.

10.11. Privacy Considerations

Several characteristics of HTTP/3 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client.

HTTP/3's preference for using a single QUIC connection allows correlation of a user's activity on a site. Reusing connections for different origins allows for correlation of activity across those origins.

Several features of QUIC solicit immediate responses and can be used by an endpoint to measure latency to their peer; this might have privacy implications in certain scenarios.

11. IANA Considerations

This document registers a new ALPN protocol ID (Section 11.1) and creates new registries that manage the assignment of codepoints in HTTP/3.

11.1. Registration of HTTP/3 Identification String

This document creates a new registration for the identification of HTTP/3 in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "h3" string identifies HTTP/3:

Protocol: HTTP/3

Identification Sequence: 0x68 0x33 ("h3")

Specification: This document

11.2. New Registries

New registries created in this document operate under the QUIC registration policy documented in Section 22.1 of [QUIC-TRANSPORT]. These registries all include the common set of fields listed in Section 22.1.1 of [QUIC-TRANSPORT]. These registries [SHALL be/are] collected under a "Hypertext Transfer Protocol version 3 (HTTP/3) Parameters" heading.

The initial allocations in these registries created in this document are all assigned permanent status and list a change controller of the IETF and a contact of the HTTP working group (ietf-http-wg@w3.org).

11.2.1. Frame Types

This document establishes a registry for HTTP/3 frame type codes. The "HTTP/3 Frame Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [HTTP2], it is preferable that the assignments parallel each other where the code spaces overlap. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following field:

Frame Type: A name or label for the frame type.

Specifications of frame types MUST include a description of the frame layout and its semantics, including any parts of the frame that are conditionally present.

The entries in Table 2 are registered by this document.

Frame Type	Value	Specification
DATA	0x0	Section 7.2.1
HEADERS	0x1	Section 7.2.2
Reserved	0x2	N/A
CANCEL_PUSH	0x3	Section 7.2.3
SETTINGS	0x4	Section 7.2.4
PUSH_PROMISE	0x5	Section 7.2.5
Reserved	0x6	N/A
GOAWAY	0x7	Section 7.2.6
Reserved	0x8	N/A
Reserved	0x9	N/A
MAX_PUSH_ID	0xd	Section 7.2.7

Table 2: Initial HTTP/3 Frame Types

Each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

11.2.2. Settings Parameters

This document establishes a registry for HTTP/3 settings. The "HTTP/3 Settings" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Settings" registry defined in [HTTP2], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

Setting Name: A symbolic name for the setting. Specifying a setting name is optional.

Default: The value of the setting unless otherwise indicated. A default SHOULD be the most restrictive possible value.

The entries in Table 3 are registered by this document.

Setting Name	Value	Specification	Default
Reserved	0x0	N/A	N/A
Reserved	0x2	N/A	N/A
Reserved	0x3	N/A	N/A
Reserved	0x4	N/A	N/A
Reserved	0x5	N/A	N/A
MAX_FIELD_SECTION_SIZE	0x6	Section 7.2.4.1	Unlimited

Table 3: Initial HTTP/3 Settings

Each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

11.2.3. Error Codes

This document establishes a registry for HTTP/3 error codes. The "HTTP/3 Error Code" registry manages a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated. Use of values that are registered in the "HTTP/2 Error Code" registry is discouraged, and expert reviewers MAY reject such registrations.

In addition to common fields as described in Section 11.2, this registry includes two additional fields. Permanent registrations in this registry MUST include the following field:

Name: A name for the error code.

Description: A brief description of the error code semantics.

The entries in Table 4 are registered by this document. These error codes were selected from the range that operates on a Specification Required policy to avoid collisions with HTTP/2 error codes.

Name	Value	Description	Specification
H3_NO_ERROR	0x100	No error	Section 8.1
H3_GENERAL_PROTOCOL_ERROR	0x101	General protocol error	Section 8.1
H3_INTERNAL_ERROR	0x102	Internal error	Section 8.1
H3_STREAM_CREATION_ERROR	0x103	Stream	Section 8.1

		creation error	
H3_CLOSED_CRITICAL_STREAM	0x104	Critical stream was closed	Section 8.1
H3_FRAME_UNEXPECTED	0x105	Frame not permitted in the current state	Section 8.1
H3_FRAME_ERROR	0x106	Frame violated layout or size rules	Section 8.1
H3_EXCESSIVE_LOAD	0x107	Peer generating excessive load	Section 8.1
H3_ID_ERROR	0x108	An identifier was used incorrectly	Section 8.1
H3_SETTINGS_ERROR	0x109	SETTINGS frame contained invalid values	Section 8.1
H3_MISSING_SETTINGS	0x10a	No SETTINGS frame received	Section 8.1
H3_REQUEST_REJECTED	0x10b	Request not processed	Section 8.1
H3_REQUEST_CANCELLED	0x10c	Data no longer needed	Section 8.1
H3_REQUEST_INCOMPLETE	0x10d	Stream terminated early	Section 8.1

H3_MESSAGE_ERROR	0x10e	Malformed message	Section 8.1
H3_CONNECT_ERROR	0x10f	TCP reset or error on CONNECT request	Section 8.1
H3_VERSION_FALLBACK	0x110	Retry over HTTP/1.1	Section 8.1

Table 4: Initial HTTP/3 Error Codes

Each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

11.2.4. Stream Types

This document establishes a registry for HTTP/3 unidirectional stream types. The "HTTP/3 Stream Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

Stream Type: A name or label for the stream type.

Sender: Which endpoint on an HTTP/3 connection may initiate a stream of this type. Values are "Client", "Server", or "Both".

Specifications for permanent registrations MUST include a description of the stream type, including the layout and semantics of the stream contents.

The entries in the following table are registered by this document.

Stream Type	Value	Specification	Sender
Control Stream	0x00	Section 6.2.1	Both
Push Stream	0x01	Section 4.4	Server

Table 5

Each code of the format "0x1f * N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

12. References

12.1. Normative References

- [ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.
- [CACHING] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-14.txt>>.
- [HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/info/rfc8470>>.
- [QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-21, 2 February 2021, <<https://tools.ietf.org/html/draft-ietf-quic-qpack-21>>.
- [QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 2 February 2021, <<https://tools.ietf.org/html/draft-ietf-quic-transport-34>>.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [SEMANTICS] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-semantics-14.txt>>.
- [URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.

12.2. Informative References

- [BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <<http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>>.

[DNS-TERMS]

Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <<https://www.rfc-editor.org/info/rfc8499>>.

[HPACK]

Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<https://www.rfc-editor.org/info/rfc7541>>.

[HTTP11]

Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-messaging-14.txt>>.

[HTTP2]

Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

[RFC6585]

Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <<https://www.rfc-editor.org/info/rfc6585>>.

[RFC8164]

Nottingham, M. and M. Thomson, "Opportunistic Security for HTTP/2", RFC 8164, DOI 10.17487/RFC8164, May 2017, <<https://www.rfc-editor.org/info/rfc8164>>.

[TFO]

Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

[TLS13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

Appendix A. Considerations for Transitioning from HTTP/2

HTTP/3 is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/3, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/3.

HTTP/3 begins from the premise that similarity to HTTP/2 is preferable, but not a hard requirement. HTTP/3 departs from HTTP/2 where QUIC differs from TCP, either to take advantage of QUIC features (like streams) or to accommodate important shortcomings (such as a lack of total ordering). These differences make HTTP/3 similar to HTTP/2 in key aspects, such as the relationship of requests and responses to streams. However, the details of the HTTP/3 design are substantially different from HTTP/2.

Some important departures are noted in this section.

A.1. Streams

HTTP/3 permits use of a larger number of streams ($2^{62}-1$) than HTTP/2. The same considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

In contrast to HTTP/2, stream concurrency in HTTP/3 is managed by QUIC. QUIC considers a stream closed when all data has been received and sent data has been acknowledged by the peer. HTTP/2 considers a stream closed when the frame containing the END_STREAM bit has been committed to the transport. As a result, the stream for an equivalent exchange could remain "active" for a longer period of time. HTTP/3 servers might choose to permit a larger number of concurrent client-initiated bidirectional streams to achieve equivalent concurrency to HTTP/2, depending on the expected usage patterns.

In HTTP/2, only request and response bodies (the frame payload of DATA frames) are subject to flow control. All HTTP/3 frames are sent on QUIC streams, so all frames on all streams are flow-controlled in HTTP/3.

Due to the presence of other unidirectional stream types, HTTP/3 does not rely exclusively on the number of concurrent unidirectional streams to control the number of concurrent in-flight pushes. Instead, HTTP/3 clients use the MAX_PUSH_ID frame to control the number of pushes received from an HTTP/3 server.

A.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an `END_STREAM` flag is not required. This permits the removal of the `Flags` field from the generic frame layout.

Frame payloads are largely drawn from [HTTP2]. However, QUIC includes many features (e.g., flow control) that are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/3. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/3 implementations. However, even frame types that appear in both mappings do not have identical semantics.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/3 will break them.

Some examples of feature adaptations are described below, as well as general guidance to extension frame implementors converting an HTTP/2 extension to HTTP/3.

A.2.1. Prioritization Differences

HTTP/2 specifies priority assignments in `PRIORITY` frames and (optionally) in `HEADERS` frames. HTTP/3 does not provide a means of signaling priority.

Note that while there is no explicit signaling for priority, this does not mean that prioritization is not important for achieving good performance.

A.2.2. Field Compression Differences

HPACK was designed with the assumption of in-order delivery. A sequence of encoded field sections must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

Because this total ordering is not provided by QUIC, HTTP/3 uses a modified version of HPACK, called QPACK. QPACK uses a single unidirectional stream to make all modifications to the dynamic table, ensuring a total order of updates. All frames that contain encoded fields merely reference the table state at a given time without modifying it.

[QPACK] provides additional details.

A.2.3. Flow Control Differences

HTTP/2 specifies a stream flow control mechanism. Although all HTTP/2 frames are delivered on streams, only the DATA frame payload is subject to flow control. QUIC provides flow control for stream data and all HTTP/3 frame types defined in this document are sent on streams. Therefore, all frame headers and payload are subject to flow control.

A.2.4. Guidance for New Frame Type Definitions

Frame type definitions in HTTP/3 often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allows for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/3 use an identifier other than a Stream ID (e.g., Push IDs). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Because the Flags field is not present in generic HTTP/3 frames, those frames that depend on the presence of flags need to allocate space for flags as part of their frame payload.

Other than these issues, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with a control stream in HTTP/3. HTTP/3 extensions will not assume ordering, but would not be harmed by ordering, and are expected to be portable to HTTP/2.

A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types

DATA (0x0): Padding is not defined in HTTP/3 frames. See Section 7.2.1.

HEADERS (0x1): The PRIORITY region of HEADERS is not defined in HTTP/3 frames. Padding is not defined in HTTP/3 frames. See Section 7.2.2.

PRIORITY (0x2): As described in Appendix A.2.1, HTTP/3 does not

provide a means of signaling priority.

RST_STREAM (0x3): RST_STREAM frames do not exist in HTTP/3, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL_PUSH frame (Section 7.2.3).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 7.2.4 and Appendix A.3.

PUSH_PROMISE (0x5): The PUSH_PROMISE frame does not reference a stream; instead the push stream references the PUSH_PROMISE frame using a Push ID. See Section 7.2.5.

PING (0x6): PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY does not contain an error code. In the client to server direction, it carries a Push ID instead of a server initiated stream ID. See Section 7.2.6.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist in HTTP/3, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist in HTTP/3; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/3 if still applicable. The IDs of frames defined in [HTTP2] have been reserved for simplicity. Note that the frame type space in HTTP/3 is substantially larger (62 bits versus 8 bits), so many HTTP/3 frame types have no equivalent HTTP/2 code points. See Section 11.2.1.

A.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, as the first frame of the control stream, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/3. The HTTP-level setting that is retained in HTTP/3 has the same value as in HTTP/2. The superseded settings are reserved, and their receipt is an error. See Section 7.2.4.1 for discussion of both the retained and reserved values.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE (0x1): See [QPACK].

SETTINGS_ENABLE_PUSH (0x2): This is removed in favor of the MAX_PUSH_ID frame, which provides a more granular control over server push. Specifying a setting with the identifier 0x2 (corresponding to the SETTINGS_ENABLE_PUSH parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_CONCURRENT_STREAMS (0x3): QUIC controls the largest open Stream ID as part of its flow control logic. Specifying a setting with the identifier 0x3 (corresponding to the SETTINGS_MAX_CONCURRENT_STREAMS parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE (0x4): QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying a setting with the identifier 0x4 (corresponding to the SETTINGS_INITIAL_WINDOW_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE (0x5): This setting has no equivalent in HTTP/3. Specifying a setting with the identifier 0x5 (corresponding to the SETTINGS_MAX_FRAME_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): This setting identifier has been renamed SETTINGS_MAX_FIELD_SECTION_SIZE.

In HTTP/3, setting values are variable-length integers (6, 14, 30, or 62 bits long) rather than fixed-length 32-bit fields as in HTTP/2. This will often produce a shorter encoding, but can produce a longer encoding for settings that use the full 32-bit space. Settings ported from HTTP/2 might choose to redefine their value to limit it to 30 bits for more efficient encoding, or to make use of the 62-bit space if more than 30 bits are required.

Settings need to be defined separately for HTTP/2 and HTTP/3. The IDs of settings defined in [HTTP2] have been reserved for simplicity. Note that the settings identifier space in HTTP/3 is substantially larger (62 bits versus 16 bits), so many HTTP/3 settings have no equivalent HTTP/2 code point. See Section 11.2.2.

As QUIC streams might arrive out of order, endpoints are advised not to wait for the peers' settings to arrive before responding to other streams. See Section 7.2.4.2.

A.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, the differences between HTTP/2 and HTTP/3 mean that error codes are not directly portable between versions.

The HTTP/2 error codes defined in Section 7 of [HTTP2] logically map to the HTTP/3 error codes as follows:

NO_ERROR (0x0): H3_NO_ERROR in Section 8.1.

PROTOCOL_ERROR (0x1): This is mapped to H3_GENERAL_PROTOCOL_ERROR except in cases where more specific error codes have been defined. Such cases include H3_FRAME_UNEXPECTED, H3_MESSAGE_ERROR, and H3_CLOSED_CRITICAL_STREAM defined in Section 8.1.

INTERNAL_ERROR (0x2): H3_INTERNAL_ERROR in Section 8.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgment of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management.

FRAME_SIZE_ERROR (0x6): H3_FRAME_ERROR error code defined in Section 8.1.

REFUSED_STREAM (0x7): H3_REQUEST_REJECTED (in Section 8.1) is used to indicate that a request was not processed. Otherwise, not applicable because QUIC handles stream management.

CANCEL (0x8): H3_REQUEST_CANCELLED in Section 8.1.

COMPRESSION_ERROR (0x9): Multiple error codes are defined in [QPACK].

CONNECT_ERROR (0xa): H3_CONNECT_ERROR in Section 8.1.

ENHANCE_YOUR_CALM (0xb): H3_EXCESSIVE_LOAD in Section 8.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): H3_VERSION_FALLBACK in Section 8.1.

Error codes need to be defined for HTTP/2 and HTTP/3 separately. See Section 11.2.3.

A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors

An intermediary that converts between HTTP/2 and HTTP/3 may encounter error conditions from either upstream. It is useful to communicate the occurrence of error to the downstream but error codes largely reflect connection-local problems that generally do not make sense to propagate.

An intermediary that encounters an error from an upstream origin can indicate this by sending an HTTP status code such as 502, which is suitable for a broad class of errors.

There are some rare cases where it is beneficial to propagate the error by mapping it to the closest matching error type to the receiver. For example, an intermediary that receives an HTTP/2 stream error of type `REFUSED_STREAM` from the origin has a clear signal that the request was not processed and that the request is safe to retry. Propagating this error condition to the client as an HTTP/3 stream error of type `H3_REQUEST_REJECTED` allows the client to take the action it deems most appropriate. In the reverse direction, the intermediary might deem it beneficial to pass on client request cancellations that are indicated by terminating a stream with `H3_REQUEST_CANCELLED`; see Section 4.1.2.

Conversion between errors is described in the logical mapping. The error codes are defined in non-overlapping spaces in order to protect against accidental conversion that could result in the use of inappropriate or unknown error codes for the target version. An intermediary is permitted to promote stream errors to connection errors but they should be aware of the cost to the HTTP/3 connection for what might be a temporary or intermittent error.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-http-32

- * Removed draft version guidance; added final version string
- * Added `H3_MESSAGE_ERROR` for malformed messages

B.2. Since draft-ietf-quic-http-31

Editorial changes only.

B.3. Since draft-ietf-quic-http-30

Editorial changes only.

B.4. Since draft-ietf-quic-http-29

- * Require a connection error if a reserved frame type that corresponds to a frame in HTTP/2 is received (#3991, #3993)
- * Require a connection error if a reserved setting that corresponds to a setting in HTTP/2 is received (#3954, #3955)

B.5. Since draft-ietf-quic-http-28

- * CANCEL_PUSH is recommended even when the stream is reset (#3698, #3700)
- * Use H3_ID_ERROR when GOAWAY contains a larger identifier (#3631, #3634)

B.6. Since draft-ietf-quic-http-27

- * Updated text to refer to latest HTTP revisions
- * Use the HTTP definition of authority for establishing and coalescing connections (#253, #2223, #3558)
- * Define use of GOAWAY from both endpoints (#2632, #3129)
- * Require either :authority or Host if the URI scheme has a mandatory authority component (#3408, #3475)

B.7. Since draft-ietf-quic-http-26

- * No changes

B.8. Since draft-ietf-quic-http-25

- * Require QUICv1 for HTTP/3 (#3117, #3323)
- * Remove DUPLICATE_PUSH and allow duplicate PUSH_PROMISE (#3275, #3309)
- * Clarify the definition of "malformed" (#3352, #3345)

B.9. Since draft-ietf-quic-http-24

- * Removed H3_EARLY_RESPONSE error code; H3_NO_ERROR is recommended instead (#3130, #3208)
- * Unknown error codes are equivalent to H3_NO_ERROR (#3276, #3331)
- * Some error codes are reserved for greasing (#3325, #3360)

B.10. Since draft-ietf-quic-http-23

- * Removed "quic" Alt-Svc parameter (#3061, #3118)
- * Clients need not persist unknown settings for use in 0-RTT (#3110, #3113)
- * Clarify error cases around CANCEL_PUSH (#2819, #3083)

B.11. Since draft-ietf-quic-http-22

- * Removed priority signaling (#2922, #2924)
- * Further changes to error codes (#2662, #2551):
 - Error codes renumbered
 - HTTP_MALFORMED_FRAME replaced by HTTP_FRAME_ERROR, HTTP_ID_ERROR, and others
- * Clarify how unknown frame types interact with required frame sequence (#2867, #2858)
- * Describe interactions with the transport in terms of defined interface terms (#2857, #2805)
- * Require the use of the "http-opportunistic" resource (RFC 8164) when scheme is "http" (#2439, #2973)
- * Settings identifiers cannot be duplicated (#2979)
- * Changes to SETTINGS frames in 0-RTT (#2972, #2790, #2945):
 - Servers must send all settings with non-default values in their SETTINGS frame, even when resuming
 - If a client doesn't have settings associated with a 0-RTT ticket, it uses the defaults

- Servers can't accept early data if they cannot recover the settings the client will have remembered
- * Clarify that Upgrade and the 101 status code are prohibited (#2898, #2889)
- * Clarify that frame types reserved for greasing can occur on any stream, but frame types reserved due to HTTP/2 correspondence are prohibited (#2997, #2692, #2693)
- * Unknown error codes cannot be treated as errors (#2998, #2816)

B.12. Since draft-ietf-quic-http-21

No changes

B.13. Since draft-ietf-quic-http-20

- * Prohibit closing the control stream (#2509, #2666)
- * Change default priority to use an orphan node (#2502, #2690)
- * Exclusive priorities are restored (#2754, #2781)
- * Restrict use of frames when using CONNECT (#2229, #2702)
- * Close and maybe reset streams if a connection error occurs for CONNECT (#2228, #2703)
- * Encourage provision of sufficient unidirectional streams for QPACK (#2100, #2529, #2762)
- * Allow extensions to use server-initiated bidirectional streams (#2711, #2773)
- * Clarify use of maximum header list size setting (#2516, #2774)
- * Extensive changes to error codes and conditions of their sending
 - Require connection errors for more error conditions (#2511, #2510)
 - Updated the error codes for illegal GOAWAY frames (#2714, #2707)
 - Specified error code for HEADERS on control stream (#2708)
 - Specified error code for servers receiving PUSH_PROMISE (#2709)

- Specified error code for receiving DATA before HEADERS (#2715)
- Describe malformed messages and their handling (#2410, #2764)
- Remove HTTP_PUSH_ALREADY_IN_CACHE error (#2812, #2813)
- Refactor Push ID related errors (#2818, #2820)
- Rationalize HTTP/3 stream creation errors (#2821, #2822)

B.14. Since draft-ietf-quic-http-19

- * SETTINGS_NUM_PLACEHOLDERS is 0x9 (#2443, #2530)
- * Non-zero bits in the Empty field of the PRIORITY frame MAY be treated as an error (#2501)

B.15. Since draft-ietf-quic-http-18

- * Resetting streams following a GOAWAY is recommended, but not required (#2256, #2457)
- * Use variable-length integers throughout (#2437, #2233, #2253, #2275)
 - Variable-length frame types, stream types, and settings identifiers
 - Renumbered stream type assignments
 - Modified associated reserved values
- * Frame layout switched from Length-Type-Value to Type-Length-Value (#2395, #2235)
- * Specified error code for servers receiving DUPLICATE_PUSH (#2497)
- * Use connection error for invalid PRIORITY (#2507, #2508)

B.16. Since draft-ietf-quic-http-17

- * HTTP_REQUEST_REJECTED is used to indicate a request can be retried (#2106, #2325)
- * Changed error code for GOAWAY on the wrong stream (#2231, #2343)

B.17. Since draft-ietf-quic-http-16

- * Rename "HTTP/QUIC" to "HTTP/3" (#1973)
- * Changes to PRIORITY frame (#1865, #2075)
 - Permitted as first frame of request streams
 - Remove exclusive reprioritization
 - Changes to Prioritized Element Type bits
- * Define DUPLICATE_PUSH frame to refer to another PUSH_PROMISE (#2072)
- * Set defaults for settings, allow request before receiving SETTINGS (#1809, #1846, #2038)
- * Clarify message processing rules for streams that aren't closed (#1972, #2003)
- * Removed reservation of error code 0 and moved HTTP_NO_ERROR to this value (#1922)
- * Removed prohibition of zero-length DATA frames (#2098)

B.18. Since draft-ietf-quic-http-15

Substantial editorial reorganization; no technical changes.

B.19. Since draft-ietf-quic-http-14

- * Recommend sensible values for QUIC transport parameters (#1720, #1806)
- * Define error for missing SETTINGS frame (#1697, #1808)
- * Setting values are variable-length integers (#1556, #1807) and do not have separate maximum values (#1820)
- * Expanded discussion of connection closure (#1599, #1717, #1712)
- * HTTP_VERSION_FALLBACK falls back to HTTP/1.1 (#1677, #1685)

B.20. Since draft-ietf-quic-http-13

- * Reserved some frame types for grease (#1333, #1446)

- * Unknown unidirectional stream types are tolerated, not errors; some reserved for grease (#1490, #1525)
- * Require settings to be remembered for 0-RTT, prohibit reductions (#1541, #1641)
- * Specify behavior for truncated requests (#1596, #1643)

B.21. Since draft-ietf-quic-http-12

- * TLS SNI extension isn't mandatory if an alternative method is used (#1459, #1462, #1466)
- * Removed flags from HTTP/3 frames (#1388, #1398)
- * Reserved frame types and settings for use in preserving extensibility (#1333, #1446)
- * Added general error code (#1391, #1397)
- * Unidirectional streams carry a type byte and are extensible (#910, #1359)
- * Priority mechanism now uses explicit placeholders to enable persistent structure in the tree (#441, #1421, #1422)

B.22. Since draft-ietf-quic-http-11

- * Moved QPACK table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

B.23. Since draft-ietf-quic-http-10

- * Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.24. Since draft-ietf-quic-http-09

- * Selected QCRAM for header compression (#228, #1117)
- * The server_name TLS extension is now mandatory (#296, #495)
- * Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.25. Since draft-ietf-quic-http-08

- * Clarified connection coalescing rules (#940, #1024)

B.26. Since draft-ietf-quic-http-07

- * Changes for integer encodings in QUIC (#595, #905)
- * Use unidirectional streams as appropriate (#515, #240, #281, #886)
- * Improvement to the description of GOAWAY (#604, #898)
- * Improve description of server push usage (#947, #950, #957)

B.27. Since draft-ietf-quic-http-06

- * Track changes in QUIC error code usage (#485)

B.28. Since draft-ietf-quic-http-05

- * Made push ID sequential, add MAX_PUSH_ID, remove SETTINGS_ENABLE_PUSH (#709)
- * Guidance about keep-alive and QUIC PINGs (#729)
- * Expanded text on GOAWAY and cancellation (#757)

B.29. Since draft-ietf-quic-http-04

- * Cite RFC 5234 (#404)
- * Return to a single stream per request (#245, #557)
- * Use separate frame type and settings registries from HTTP/2 (#81)
- * SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)
- * Restored GOAWAY (#696)
- * Identify server push using Push ID rather than a stream ID (#702, #281)
- * DATA frames cannot be empty (#700)

B.30. Since draft-ietf-quic-http-03

None.

B.31. Since draft-ietf-quic-http-02

- * Track changes in transport draft

B.32. Since draft-ietf-quic-http-01

- * SETTINGS changes (#181):
 - SETTINGS can be sent only once at the start of a connection; no changes thereafter
 - SETTINGS_ACK removed
 - Settings can only occur in the SETTINGS frame a single time
 - Boolean format updated
- * Alt-Svc parameter changed from "v" to "quic"; format updated (#229)
- * Closing the connection control stream or any message control stream is a fatal error (#176)
- * HPACK Sequence counter can wrap (#173)
- * 0-RTT guidance added
- * Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.33. Since draft-ietf-quic-http-00

- * Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)
- * Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)
- * Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01
- * Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)
- * Described CONNECT pseudo-method (#95)
- * Updated ALPN token and Alt-Svc guidance (#13,#87)
- * Application-layer-defined error codes (#19,#74)

B.34. Since draft-shade-quic-http2-mapping-00

- * Adopted as base for draft-ietf-quic-http
- * Updated authors/editors list

Acknowledgments

The original authors of this specification were Robbie Shade and Mike Warres.

The IETF QUIC Working Group received an enormous amount of support from many people. Among others, the following people provided substantial contributions to this document:

- * Bence Beky
- * Daan De Meyer
- * Martin Duke
- * Roy Fielding
- * Alan Frindell
- * Alessandro Ghedini
- * Nick Harper
- * Ryan Hamilton
- * Christian Huitema
- * Subodh Iyengar
- * Robin Marx
- * Patrick McManus
- * Luca Niccolini
- * (Kazuho Oku)
- * Lucas Pardue
- * Roberto Peon
- * Julian Reschke

- * Eric Rescorla
- * Martin Seemann
- * Ben Schwartz
- * Ian Swett
- * Willy Taureau
- * Martin Thomson
- * Dmitri Tikhonov
- * Tatsuhiro Tsujikawa

A portion of Mike's contribution was supported by Microsoft during his employment there.

Author's Address

Mike Bishop (editor)
Akamai

Email: mbishop@evequefou.be

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

J. Iyengar, Ed.
I. Swett, Ed.
Google
March 13, 2017

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-02

Abstract

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. QUIC implements the spirit of known TCP loss detection mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC loss detection and congestion control, and attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and TCP implementations.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at <https://github.com/quicwg> ; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/recovery> .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
2. Design of the QUIC Transmission Machinery	3
2.1. Relevant Differences Between QUIC and TCP	4
2.1.1. Monotonically Increasing Packet Numbers	4
2.1.2. No Reneging	4
2.1.3. More ACK Ranges	5
2.1.4. Explicit Correction For Delayed Acks	5
3. Loss Detection	5
3.1. Constants of interest	5
3.2. Variables of interest	6
3.3. Initialization	7
3.4. On Sending a Packet	7
3.5. On Ack Receipt	8
3.6. On Packet Acknowledgment	8
3.7. Setting the Loss Detection Alarm	9
3.7.1. Handshake Packets	9
3.7.2. Tail Loss Probe and Retransmission Timeout	9
3.7.3. Early Retransmit	9
3.7.4. Pseudocode	10
3.8. On Alarm Firing	10
3.9. Detecting Lost Packets	11
3.9.1. Handshake Packets	11
3.9.2. Pseudocode	11
4. Congestion Control	12
5. IANA Considerations	12
6. References	12
6.1. Normative References	12
6.2. Informative References	13
Appendix A. Acknowledgments	13
Appendix B. Change Log	13

B.1. Since draft-ietf-quic-recovery-01	14
B.2. Since draft-ietf-quic-recovery-00:	14
B.3. Since draft-iyengar-quic-loss-recovery-01:	14
Authors' Addresses	14

1. Introduction

QUIC is a new multiplexed and secure transport atop UDP. QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. The QUIC protocol is described in [QUIC-TRANSPORT].

QUIC implements the spirit of known TCP loss recovery mechanisms, described in RFCs, various Internet-drafts, and also those prevalent in the Linux TCP implementation. This document describes QUIC congestion control and loss recovery, and where applicable, attributes the TCP equivalent in RFCs, Internet-drafts, academic papers, and/or TCP implementations.

This document first describes pre-requisite parts of the QUIC transmission machinery, then discusses QUIC's default congestion control and loss detection mechanisms, and finally lists the various TCP mechanisms that QUIC loss detection implements (in spirit.)

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

2. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which includes a packet sequence number (referred to below as a packet number). These packet numbers never repeat in the lifetime of a connection, and are monotonically increasing, which makes duplicate detection trivial. This fundamental design decision obviates the need for disambiguating between transmissions and retransmissions and eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

Every packet may contain several frames. We outline the frames that are important to the loss detection and congestion control machinery below.

- o Retransmittable frames are frames requiring reliable delivery. The most common are STREAM frames, which typically contain application data.

- o Crypto handshake data is also sent as STREAM data, and uses the reliability machinery of QUIC underneath.
- o ACK frames contain acknowledgment information. QUIC uses a SACK-based scheme, where acks express up to 256 ranges. The ACK frame also includes a receive timestamp for each packet newly acked.

2.1. Relevant Differences Between QUIC and TCP

There are some notable differences between QUIC and TCP which are important for reasoning about the differences between the loss recovery mechanisms employed by the two protocols. We briefly describe these differences below.

2.1.1. Monotonically Increasing Packet Numbers

TCP conflates transmission sequence number at the sender with delivery sequence number at the receiver, which results in retransmissions of the same data carrying the same sequence number, and consequently to problems caused by "retransmission ambiguity". QUIC separates the two: QUIC uses a packet sequence number (referred to as the "packet number") for transmissions, and any data that is to be delivered to the receiving application(s) is sent in one or more streams, with stream offsets encoded within STREAM frames inside of packets that determine delivery order.

QUIC's packet number is strictly increasing, and directly encodes transmission order. A higher QUIC packet number signifies that the packet was sent later, and a lower QUIC packet number signifies that the packet was sent earlier. When a packet containing frames is deemed lost, QUIC rebundles necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers - a non-trivial task, especially when TCP timestamps are not available.

2.1.2. No Reneging

QUIC ACKs contain information that is equivalent to TCP SACK, but QUIC does not allow any acked packet to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

2.1.3. More ACK Ranges

QUIC supports up to 256 ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery.

2.1.4. Explicit Correction For Delayed Acks

QUIC ACKs explicitly encode the delay incurred at the receiver between when a packet is received and when the corresponding ACK is sent. This allows the receiver of the ACK to adjust for receiver delays, specifically the delayed ack timer, when estimating the path RTT. This mechanism also allows a receiver to measure and report the delay from when a packet was received by the OS kernel, which is useful in receivers which may incur delays such as context-switch latency before a userspace QUIC receiver processes a received packet.

3. Loss Detection

We now describe QUIC's loss detection as functions that should be called on packet transmission, when a packet is acked, and timer expiration events.

3.1. Constants of interest

Constants used in loss recovery and congestion control are based on a combination of RFCs, papers, and common practice. Some may need to be changed or negotiated in order to better suit a variety of environments.

kMaxTLPs (default 2): Maximum number of tail loss probes before an RTO fires.

kReorderingThreshold (default 3): Maximum reordering in packet number space before FACK style loss detection considers a packet lost.

kTimeReorderingFraction (default 1/8): Maximum reordering in time space before time based loss detection considers a packet lost. In fraction of an RTT.

kMinTLPTimeout (default 10ms): Minimum time in the future a tail loss probe alarm may be set for.

kMinRTOTimeout (default 200ms): Minimum time in the future an RTO alarm may be set for.

kDelayedAckTimeout (default 25ms): The length of the peer's delayed ack timer.

kDefaultInitialRtt (default 100ms): The default RTT used before an RTT sample is taken.

3.2. Variables of interest

We first describe the variables required to implement the loss detection mechanisms described in this section.

loss_detection_alarm: Multi-modal alarm used for loss detection.

handshake_count: The number of times the handshake packets have been retransmitted without receiving an ack.

tlp_count: The number of times a tail loss probe has been sent without receiving an ack.

rto_count: The number of times an rto has been sent without receiving an ack.

smoothed_rtt: The smoothed RTT of the connection, computed as described in [RFC6298]

rttvar: The RTT variance, computed as described in [RFC6298]

initial_rtt: The initial RTT used before any RTT measurements have been made.

reordering_threshold: The largest delta between the largest acked retransmittable packet and a packet containing retransmittable frames before it's declared lost.

time_reordering_fraction: The reordering window as a fraction of $\max(\text{smoothed_rtt}, \text{latest_rtt})$.

loss_time: The time at which the next packet will be considered lost based on early transmit or exceeding the reordering window in time.

sent_packets: An association of packet numbers to information about them, including a number field indicating the packet number, a time field indicating the time a packet was sent, and a bytes field indicating the packet's size. sent_packets is ordered by packet number, and packets remain in sent_packets until acknowledged or lost.

3.3. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
loss_detection_alarm.reset()
handshake_count = 0
tlp_count = 0
rto_count = 0
if (UsingTimeLossDetection())
    reordering_threshold = infinite
    time_reordering_fraction = kTimeReorderingFraction
else:
    reordering_threshold = kReorderingThreshold
    time_reordering_fraction = infinite
loss_time = 0
smoothed_rtt = 0
rttvar = 0
initial_rtt = kDefaultInitialRtt
```

3.4. On Sending a Packet

After any packet is sent, be it a new transmission or a rebundled transmission, the following OnPacketSent function is called. The parameters to OnPacketSent are as follows:

- o packet_number: The packet number of the sent packet.
- o is_retransmittable: A boolean that indicates whether the packet contains at least one frame requiring reliable deliver. The retransmittability of various QUIC frames is described in [QUIC-TRANSPORT]. If false, it is still acceptable for an ack to be received for this packet. However, a caller MUST NOT set is_retransmittable to true if an ack is not expected.
- o sent_bytes: The number of bytes sent in the packet.

Pseudocode for OnPacketSent follows:

```
OnPacketSent(packet_number, is_retransmittable, sent_bytes):
    sent_packets[packet_number].packet_number = packet_number
    sent_packets[packet_number].time = now
    if is_retransmittable:
        sent_packets[packet_number].bytes = sent_bytes
        SetLossDetectionAlarm()
```

3.5. On Ack Receipt

When an ack is received, it may acknowledge 0 or more packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
OnAckReceived(ack):
    // If the largest acked is newly acked, update the RTT.
    if (sent_packets[ack.largest_acked]):
        rtt_sample = now - sent_packets[ack.largest_acked].time
        if (rtt_sample > ack.ack_delay):
            rtt_sample -= ack.delay
        UpdateRtt(rtt_sample)
    // Find all newly acked packets.
    for acked_packet_number in DetermineNewlyAkedPackets():
        OnPacketAked(acked_packet_number)

    DetectLostPackets(ack.largest_acked_packet)
    SetLossDetectionAlarm()

UpdateRtt(rtt_sample):
    // Based on {{RFC6298}}.
    if (smoothed_rtt == 0):
        smoothed_rtt = rtt_sample
        rttvar = rtt_sample / 2
    else:
        rttvar = 3/4 * rttvar + 1/4 * (smoothed_rtt - rtt_sample)
        smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * rtt_sample
```

3.6. On Packet Acknowledgment

When a packet is acked for the first time, the following OnPacketAked function is called. Note that a single ACK frame may newly acknowledge several packets. OnPacketAked must be called once for each of these newly acked packets.

OnPacketAked takes one parameter, `acked_packet`, which is the packet number of the newly acked packet, and returns a list of packet numbers that are detected as lost.

Pseudocode for OnPacketAked follows:

```
OnPacketAked(acked_packet_number):
    handshake_count = 0
    tlp_count = 0
    rto_count = 0
    sent_packets.remove(acked_packet_number)
```

3.7. Setting the Loss Detection Alarm

QUIC loss detection uses a single alarm for all timer-based loss detection. The duration of the alarm is based on the alarm's mode, which is set in the packet and timer events further below. The function `SetLossDetectionAlarm` defined below shows how the single timer is set based on the alarm mode.

3.7.1. Handshake Packets

The initial flight has no prior RTT sample. A client SHOULD remember the previous RTT it observed when resumption is attempted and use that for an initial RTT value. If no previous RTT is available, the initial RTT defaults to 200ms. Once an RTT measurement is taken, it MUST replace `initial_rtt`.

Endpoints MUST retransmit handshake frames if not acknowledged within a time limit. This time limit will start as the largest of twice the rtt value and `MinTLPTimeout`. Each consecutive handshake retransmission doubles the time limit, until an acknowledgement is received.

Handshake frames may be cancelled by handshake state transitions. In particular, all non-protected frames SHOULD be no longer be transmitted once packet protection is available.

When stateless rejects are in use, the connection is considered immediately closed once a reject is sent, so no timer is set to retransmit the reject.

Version negotiation packets are always stateless, and MUST be sent once per handshake packet that uses an unsupported QUIC version, and MAY be sent in response to 0RTT packets.

3.7.2. Tail Loss Probe and Retransmission Timeout

Tail loss probes [I-D.dukkipati-tcpm-tcp-loss-probe] and retransmission timeouts[RFC6298] are an alarm based mechanism to recover from cases when there are outstanding retransmittable packets, but an acknowledgement has not been received in a timely manner.

3.7.3. Early Retransmit

Early retransmit [RFC5827] is implemented with a 1/4 RTT timer. It is part of QUIC's time based loss detection, but is always enabled, even when only packet reordering loss detection is enabled.

3.7.4. Pseudocode

Pseudocode for SetLossDetectionAlarm follows:

```
SetLossDetectionAlarm():
  if (retransmittable packets are not outstanding):
    loss_detection_alarm.cancel();
    return

  if (handshake packets are outstanding):
    // Handshake retransmission alarm.
    if (smoothed_rtt == 0):
      alarm_duration = 2 * initial_rtt
    else:
      alarm_duration = 2 * smoothed_rtt
    alarm_duration = max(alarm_duration, kMinTLPTimeout)
    alarm_duration = alarm_duration << handshake_count
  else if (loss_time != 0):
    // Early retransmit timer or time loss detection.
    alarm_duration = loss_time - now
  else if (tlp_count < kMaxTLPs):
    // Tail Loss Probe
    if (retransmittable_packets_outstanding = 1):
      alarm_duration = 1.5 * smoothed_rtt + kDelayedAckTimeout
    else:
      alarm_duration = kMinTLPTimeout
    alarm_duration = max(alarm_duration, 2 * smoothed_rtt)
  else:
    // RTO alarm
    if (rto_count = 0):
      alarm_duration = smoothed_rtt + 4 * rttvar
      alarm_duration = max(alarm_duration, kMinRTOTimeout)
    else:
      alarm_duration = loss_detection_alarm.get_delay() << 1

  loss_detection_alarm.set(now + alarm_duration)
```

3.8. On Alarm Firing

QUIC uses one loss recovery alarm, which when set, can be in one of several modes. When the alarm fires, the mode determines the action to be performed.

Pseudocode for OnLossDetectionAlarm follows:

```
OnLossDetectionAlarm():
    if (handshake packets are outstanding):
        // Handshake retransmission alarm.
        RetransmitAllHandshakePackets();
        handshake_count++;
    // TODO: Clarify early retransmit and time loss.
    else if (loss_time != 0):
        // Early retransmit or Time Loss Detection
        DetectLostPackets(largest_acked_packet)
    else if (tlp_count < kMaxTLPs):
        // Tail Loss Probe.
        if (HasNewDataToSend()):
            SendOnePacketOfNewData()
        else:
            RetransmitOldestPacket()
        tlp_count++
    else:
        // RTO.
        RetransmitOldestTwoPackets()
        rto_count++

    SetLossDetectionAlarm()
```

3.9. Detecting Lost Packets

Packets in QUIC are only considered lost once a larger packet number is acknowledged. DetectLostPackets is called every time an ack is received. If the loss detection alarm fires and the loss_time is set, the previous largest acked packet is supplied.

3.9.1. Handshake Packets

The receiver MUST ignore unprotected packets that ack protected packets. The receiver MUST trust protected acks for unprotected packets, however. Aside from this, loss detection for handshake packets when an ack is processed is identical to other packets.

3.9.2. Pseudocode

DetectLostPackets takes one parameter, acked, which is the largest acked packet.

Pseudocode for DetectLostPackets follows:

```

DetectLostPackets(largest_acked):
    loss_time = 0
    lost_packets = {}
    delay_until_lost = infinite;
    if (time_reordering_fraction != infinite):
        delay_until_lost =
            (1 + time_reordering_fraction) * max(latest_rtt, smoothed_rtt)
    else if (largest_acked.packet_number == largest_sent_packet):
        // Early retransmit alarm.
        delay_until_lost = 9/8 * max(latest_rtt, smoothed_rtt)
    foreach (unacked less than largest_acked.packet_number):
        time_since_sent = now() - unacked.time_sent
        packet_delta = largest_acked.packet_number - unacked.packet_number
        if (time_since_sent > delay_until_lost):
            lost_packets.insert(unacked)
        else if (packet_delta > reordering_threshold)
            lost_packets.insert(unacked)
        else if (loss_time == 0 && delay_until_lost != infinite):
            loss_time = delay_until_lost - time_since_sent

    // Inform the congestion controller of lost packets and
    // lets it decide whether to retransmit immediately.
    OnPacketsLost(lost_packets)
    foreach (packet in lost_packets)
        sent_packets.remove(packet.packet_number)

```

4. Congestion Control

(describe NewReno-style congestion control [RFC6582] for QUIC.)
 (describe appropriate byte counting.) (define recovery based on
 packet numbers.) (describe min_rtt based hystart.) (describe how
 QUIC's F-RTO [RFC5682] delays reducing CWND.) (describe PRR
 [RFC6937])

5. IANA Considerations

This document has no IANA actions. Yet.

6. References

6.1. Normative References

[QUIC-TRANSPORT]
 Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based
 Multiplexed and Secure Transport".

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

6.2. Informative References

- [I-D.dukkipati-tcpm-tcp-loss-probe] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), February 2013.
- [RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <<http://www.rfc-editor.org/info/rfc5682>>.
- [RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <<http://www.rfc-editor.org/info/rfc5827>>.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <<http://www.rfc-editor.org/info/rfc6298>>.
- [RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <<http://www.rfc-editor.org/info/rfc6582>>.
- [RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, DOI 10.17487/RFC6937, May 2013, <<http://www.rfc-editor.org/info/rfc6937>>.

Appendix A. Acknowledgments

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

B.1. Since draft-ietf-quic-recovery-01

- o Changes initial default RTT to 100ms
- o Added time-based loss detection and fixes early retransmit
- o Clarified loss recovery for handshake packets
- o Fixed references and made TCP references informative

B.2. Since draft-ietf-quic-recovery-00:

- o Improved description of constants and ACK behavior

B.3. Since draft-iyengar-quic-loss-recovery-01:

- o Adopted as base for draft-ietf-quic-recovery.
- o Updated authors/editors list.
- o Added table of contents.

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Ian Swett (editor)
Google

Email: ianswett@google.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 19 July 2021

J. Iyengar, Ed.
Fastly
I. Swett, Ed.
Google
15 January 2021

QUIC Loss Detection and Congestion Control
draft-ietf-quic-recovery-34

Abstract

This document describes loss detection and congestion control mechanisms for QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org (<mailto:quic@ietf.org>)), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-recovery>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 July 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
2. Conventions and Definitions	4
3. Design of the QUIC Transmission Machinery	5
4. Relevant Differences Between QUIC and TCP	6
4.1. Separate Packet Number Spaces	6
4.2. Monotonically Increasing Packet Numbers	6
4.3. Clearer Loss Epoch	7
4.4. No Reneging	7
4.5. More ACK Ranges	7
4.6. Explicit Correction For Delayed Acknowledgments	7
4.7. Probe Timeout Replaces RTO and TLP	7
4.8. The Minimum Congestion Window is Two Packets	8
5. Estimating the Round-Trip Time	8
5.1. Generating RTT samples	8
5.2. Estimating min_rtt	9
5.3. Estimating smoothed_rtt and rttvar	10
6. Loss Detection	12
6.1. Acknowledgment-Based Detection	12
6.1.1. Packet Threshold	13
6.1.2. Time Threshold	13
6.2. Probe Timeout	14
6.2.1. Computing PTO	14
6.2.2. Handshakes and New Paths	16
6.2.3. Speeding Up Handshake Completion	17
6.2.4. Sending Probe Packets	18
6.3. Handling Retry Packets	19
6.4. Discarding Keys and Packet State	19
7. Congestion Control	20
7.1. Explicit Congestion Notification	20
7.2. Initial and Minimum Congestion Window	21
7.3. Congestion Control States	21
7.3.1. Slow Start	22
7.3.2. Recovery	22
7.3.3. Congestion Avoidance	23
7.4. Ignoring Loss of Undecryptable Packets	23
7.5. Probe Timeout	24
7.6. Persistent Congestion	24

7.6.1. Duration	24
7.6.2. Establishing Persistent Congestion	25
7.6.3. Example	26
7.7. Pacing	27
7.8. Under-utilizing the Congestion Window	28
8. Security Considerations	28
8.1. Loss and Congestion Signals	28
8.2. Traffic Analysis	28
8.3. Misreporting ECN Markings	28
9. IANA Considerations	29
10. References	29
10.1. Normative References	29
10.2. Informative References	30
Appendix A. Loss Recovery Pseudocode	32
A.1. Tracking Sent Packets	32
A.1.1. Sent Packet Fields	32
A.2. Constants of Interest	33
A.3. Variables of interest	33
A.4. Initialization	34
A.5. On Sending a Packet	34
A.6. On Receiving a Datagram	35
A.7. On Receiving an Acknowledgment	35
A.8. Setting the Loss Detection Timer	37
A.9. On Timeout	39
A.10. Detecting Lost Packets	39
A.11. Upon Dropping Initial or Handshake Keys	40
Appendix B. Congestion Control Pseudocode	41
B.1. Constants of interest	41
B.2. Variables of interest	41
B.3. Initialization	42
B.4. On Packet Sent	42
B.5. On Packet Acknowledgment	42
B.6. On New Congestion Event	43
B.7. Process ECN Information	44
B.8. On Packets Lost	44
B.9. Removing Discarded Packets From Bytes In Flight	44
Appendix C. Change Log	45
C.1. Since draft-ietf-quic-recovery-32	45
C.2. Since draft-ietf-quic-recovery-31	45
C.3. Since draft-ietf-quic-recovery-30	45
C.4. Since draft-ietf-quic-recovery-29	45
C.5. Since draft-ietf-quic-recovery-28	46
C.6. Since draft-ietf-quic-recovery-27	46
C.7. Since draft-ietf-quic-recovery-26	46
C.8. Since draft-ietf-quic-recovery-25	46
C.9. Since draft-ietf-quic-recovery-24	46
C.10. Since draft-ietf-quic-recovery-23	46
C.11. Since draft-ietf-quic-recovery-22	47

C.12. Since draft-ietf-quic-recovery-21	47
C.13. Since draft-ietf-quic-recovery-20	47
C.14. Since draft-ietf-quic-recovery-19	47
C.15. Since draft-ietf-quic-recovery-18	48
C.16. Since draft-ietf-quic-recovery-17	48
C.17. Since draft-ietf-quic-recovery-16	49
C.18. Since draft-ietf-quic-recovery-14	49
C.19. Since draft-ietf-quic-recovery-13	49
C.20. Since draft-ietf-quic-recovery-12	50
C.21. Since draft-ietf-quic-recovery-11	50
C.22. Since draft-ietf-quic-recovery-10	50
C.23. Since draft-ietf-quic-recovery-09	50
C.24. Since draft-ietf-quic-recovery-08	50
C.25. Since draft-ietf-quic-recovery-07	50
C.26. Since draft-ietf-quic-recovery-06	51
C.27. Since draft-ietf-quic-recovery-05	51
C.28. Since draft-ietf-quic-recovery-04	51
C.29. Since draft-ietf-quic-recovery-03	51
C.30. Since draft-ietf-quic-recovery-02	51
C.31. Since draft-ietf-quic-recovery-01	51
C.32. Since draft-ietf-quic-recovery-00	51
C.33. Since draft-iyengar-quic-loss-recovery-01	51
Appendix D. Contributors	52
Acknowledgments	52
Authors' Addresses	52

1. Introduction

QUIC is a secure general-purpose transport protocol, described in [QUIC-TRANSPORT]). This document describes loss detection and congestion control mechanisms for QUIC.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Definitions of terms that are used in this document:

Ack-eliciting frames: All frames other than ACK, PADDING, and CONNECTION_CLOSE are considered ack-eliciting.

Ack-eliciting packets: Packets that contain ack-eliciting frames elicit an ACK from the receiver within the maximum acknowledgment delay and are called ack-eliciting packets.

In-flight packets: Packets are considered in-flight when they are ack-eliciting or contain a PADDING frame, and they have been sent but are not acknowledged, declared lost, or discarded along with old keys.

3. Design of the QUIC Transmission Machinery

All transmissions in QUIC are sent with a packet-level header, which indicates the encryption level and includes a packet sequence number (referred to below as a packet number). The encryption level indicates the packet number space, as described in Section 12.3 in [QUIC-TRANSPORT]. Packet numbers never repeat within a packet number space for the lifetime of a connection. Packet numbers are sent in monotonically increasing order within a space, preventing ambiguity. It is permitted for some packet numbers to never be used, leaving intentional gaps.

This design obviates the need for disambiguating between transmissions and retransmissions; this eliminates significant complexity from QUIC's interpretation of TCP loss detection mechanisms.

QUIC packets can contain multiple frames of different types. The recovery mechanisms ensure that data and frames that need reliable delivery are acknowledged or declared lost and sent in new packets as necessary. The types of frames contained in a packet affect recovery and congestion control logic:

- * All packets are acknowledged, though packets that contain no ack-eliciting frames are only acknowledged along with ack-eliciting packets.
- * Long header packets that contain CRYPTO frames are critical to the performance of the QUIC handshake and use shorter timers for acknowledgment.
- * Packets containing frames besides ACK or CONNECTION_CLOSE frames count toward congestion control limits and are considered in-flight.
- * PADDING frames cause packets to contribute toward bytes in flight without directly causing an acknowledgment to be sent.

4. Relevant Differences Between QUIC and TCP

Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones. However, protocol differences between QUIC and TCP contribute to algorithmic differences. These protocol differences are briefly described below.

4.1. Separate Packet Number Spaces

QUIC uses separate packet number spaces for each encryption level, except 0-RTT and all generations of 1-RTT keys use the same packet number space. Separate packet number spaces ensures acknowledgment of packets sent with one level of encryption will not cause spurious retransmission of packets sent with a different encryption level. Congestion control and round-trip time (RTT) measurement are unified across packet number spaces.

4.2. Monotonically Increasing Packet Numbers

TCP conflates transmission order at the sender with delivery order at the receiver, resulting in the retransmission ambiguity problem ([RETRANSMISSION]). QUIC separates transmission order from delivery order: packet numbers indicate transmission order, and delivery order is determined by the stream offsets in STREAM frames.

QUIC's packet number is strictly increasing within a packet number space, and directly encodes transmission order. A higher packet number signifies that the packet was sent later, and a lower packet number signifies that the packet was sent earlier. When a packet containing ack-eliciting frames is detected lost, QUIC includes necessary frames in a new packet with a new packet number, removing ambiguity about which packet is acknowledged when an ACK is received. Consequently, more accurate RTT measurements can be made, spurious retransmissions are trivially detected, and mechanisms such as Fast Retransmit can be applied universally, based only on packet number.

This design point significantly simplifies loss detection mechanisms for QUIC. Most TCP mechanisms implicitly attempt to infer transmission ordering based on TCP sequence numbers - a non-trivial task, especially when TCP timestamps are not available.

4.3. Clearer Loss Epoch

QUIC starts a loss epoch when a packet is lost. The loss epoch ends when any packet sent after the start of the epoch is acknowledged. TCP waits for the gap in the sequence number space to be filled, and so if a segment is lost multiple times in a row, the loss epoch may not end for several round trips. Because both should reduce their congestion windows only once per epoch, QUIC will do it once for every round trip that experiences loss, while TCP may only do it once across multiple round trips.

4.4. No Reneging

QUIC ACK frames contain information similar to that in TCP Selective Acknowledgements (SACKs, [RFC2018]). However, QUIC does not allow a packet acknowledgement to be reneged, greatly simplifying implementations on both sides and reducing memory pressure on the sender.

4.5. More ACK Ranges

QUIC supports many ACK ranges, opposed to TCP's 3 SACK ranges. In high loss environments, this speeds recovery, reduces spurious retransmits, and ensures forward progress without relying on timeouts.

4.6. Explicit Correction For Delayed Acknowledgments

QUIC endpoints measure the delay incurred between when a packet is received and when the corresponding acknowledgment is sent, allowing a peer to maintain a more accurate round-trip time estimate; see Section 13.2 of [QUIC-TRANSPORT].

4.7. Probe Timeout Replaces RTO and TLP

QUIC uses a probe timeout (PTO; see Section 6.2), with a timer based on TCP's RTO computation; see [RFC6297]. QUIC's PTO includes the peer's maximum expected acknowledgment delay instead of using a fixed minimum timeout.

Similar to the RACK-TLP loss detection algorithm for TCP ([RACK]), QUIC does not collapse the congestion window when the PTO expires, since a single packet loss at the tail does not indicate persistent congestion. Instead, QUIC collapses the congestion window when persistent congestion is declared; see Section 7.6. In doing this, QUIC avoids unnecessary congestion window reductions, obviating the need for correcting mechanisms such as F-RTO ([RFC5682]). Since QUIC does not collapse the congestion window on a PTO expiration, a QUIC

sender is not limited from sending more in-flight packets after a PTO expiration if it still has available congestion window. This occurs when a sender is application-limited and the PTO timer expires. This is more aggressive than TCP's RTO mechanism when application-limited, but identical when not application-limited.

QUIC allows probe packets to temporarily exceed the congestion window whenever the timer expires.

4.8. The Minimum Congestion Window is Two Packets

TCP uses a minimum congestion window of one packet. However, loss of that single packet means that the sender needs to waiting for a PTO (Section 6.2) to recover, which can be much longer than a round-trip time. Sending a single ack-eliciting packet also increases the chances of incurring additional latency when a receiver delays its acknowledgment.

QUIC therefore recommends that the minimum congestion window be two packets. While this increases network load, it is considered safe, since the sender will still reduce its sending rate exponentially under persistent congestion (Section 6.2).

5. Estimating the Round-Trip Time

At a high level, an endpoint measures the time from when a packet was sent to when it is acknowledged as a round-trip time (RTT) sample. The endpoint uses RTT samples and peer-reported host delays (see Section 13.2 of [QUIC-TRANSPORT]) to generate a statistical description of the network path's RTT. An endpoint computes the following three values for each path: the minimum value over a period of time (`min_rtt`), an exponentially-weighted moving average (`smoothed_rtt`), and the mean deviation (referred to as "variation" in the rest of this document) in the observed RTT samples (`rttvar`).

5.1. Generating RTT samples

An endpoint generates an RTT sample on receiving an ACK frame that meets the following two conditions:

- * the largest acknowledged packet number is newly acknowledged, and
- * at least one of the newly acknowledged packets was ack-eliciting.

The RTT sample, `latest_rtt`, is generated as the time elapsed since the largest acknowledged packet was sent:

```
latest_rtt = ack_time - send_time_of_largest_acked
```

An RTT sample is generated using only the largest acknowledged packet in the received ACK frame. This is because a peer reports acknowledgment delays for only the largest acknowledged packet in an ACK frame. While the reported acknowledgment delay is not used by the RTT sample measurement, it is used to adjust the RTT sample in subsequent computations of `smoothed_rtt` and `rttvar` (Section 5.3).

To avoid generating multiple RTT samples for a single packet, an ACK frame SHOULD NOT be used to update RTT estimates if it does not newly acknowledge the largest acknowledged packet.

An RTT sample MUST NOT be generated on receiving an ACK frame that does not newly acknowledge at least one ack-eliciting packet. A peer usually does not send an ACK frame when only non-ack-eliciting packets are received. Therefore an ACK frame that contains acknowledgments for only non-ack-eliciting packets could include an arbitrarily large ACK Delay value. Ignoring such ACK frames avoids complications in subsequent `smoothed_rtt` and `rttvar` computations.

A sender might generate multiple RTT samples per RTT when multiple ACK frames are received within an RTT. As suggested in [RFC6298], doing so might result in inadequate history in `smoothed_rtt` and `rttvar`. Ensuring that RTT estimates retain sufficient history is an open research question.

5.2. Estimating `min_rtt`

`min_rtt` is the sender's estimate of the minimum RTT observed for a given network path over a period of time. In this document, `min_rtt` is used by loss detection to reject implausibly small `rtt` samples.

`min_rtt` MUST be set to the `latest_rtt` on the first RTT sample. `min_rtt` MUST be set to the lesser of `min_rtt` and `latest_rtt` (Section 5.1) on all other samples.

An endpoint uses only locally observed times in computing the `min_rtt` and does not adjust for acknowledgment delays reported by the peer. Doing so allows the endpoint to set a lower bound for the `smoothed_rtt` based entirely on what it observes (see Section 5.3), and limits potential underestimation due to erroneously-reported delays by the peer.

The RTT for a network path may change over time. If a path's actual RTT decreases, the `min_rtt` will adapt immediately on the first low sample. If the path's actual RTT increases however, the `min_rtt` will not adapt to it, allowing future RTT samples that are smaller than the new RTT to be included in `smoothed_rtt`.

Endpoints SHOULD set the `min_rtt` to the newest RTT sample after persistent congestion is established. This is to allow a connection to reset its estimate of `min_rtt` and `smoothed_rtt` (Section 5.3) after a disruptive network event, and because it is possible that an increase in path delay resulted in persistent congestion being incorrectly declared.

Endpoints MAY re-establish the `min_rtt` at other times in the connection, such as when traffic volume is low and an acknowledgment is received with a low acknowledgment delay. Implementations SHOULD NOT refresh the `min_rtt` value too often, since the actual minimum RTT of the path is not frequently observable.

5.3. Estimating `smoothed_rtt` and `rttvar`

`smoothed_rtt` is an exponentially-weighted moving average of an endpoint's RTT samples, and `rttvar` estimates the variation in the RTT samples using a mean variation.

The calculation of `smoothed_rtt` uses RTT samples after adjusting them for acknowledgment delays. These delays are decoded from the ACK Delay field of ACK frames as described in Section 19.3 of [QUIC-TRANSPORT].

The peer might report acknowledgment delays that are larger than the peer's `max_ack_delay` during the handshake (Section 13.2.1 of [QUIC-TRANSPORT]). To account for this, the endpoint SHOULD ignore `max_ack_delay` until the handshake is confirmed, as defined in Section 4.1.2 of [QUIC-TLS]. When they occur, these large acknowledgment delays are likely to be non-repeating and limited to the handshake. The endpoint can therefore use them without limiting them to the `max_ack_delay`, avoiding unnecessary inflation of the RTT estimate.

Note that a large acknowledgment delay can result in a substantially inflated `smoothed_rtt`, if there is either an error in the peer's reporting of the acknowledgment delay or in the endpoint's `min_rtt` estimate. Therefore, prior to handshake confirmation, an endpoint MAY ignore RTT samples if adjusting the RTT sample for acknowledgment delay causes the sample to be less than the `min_rtt`.

After the handshake is confirmed, any acknowledgment delays reported by the peer that are greater than the peer's `max_ack_delay` are attributed to unintentional but potentially repeating delays, such as scheduler latency at the peer or loss of previous acknowledgments. Excess delays could also be due to a non-compliant receiver. Therefore, these extra delays are considered effectively part of path delay and incorporated into the RTT estimate.

Therefore, when adjusting an RTT sample using peer-reported acknowledgment delays, an endpoint:

- * MAY ignore the acknowledgment delay for Initial packets, since these acknowledgments are not delayed by the peer (Section 13.2.1 of [QUIC-TRANSPORT]);
- * SHOULD ignore the peer's `max_ack_delay` until the handshake is confirmed;
- * MUST use the lesser of the acknowledgment delay and the peer's `max_ack_delay` after the handshake is confirmed; and
- * MUST NOT subtract the acknowledgment delay from the RTT sample if the resulting value is smaller than the `min_rtt`. This limits the underestimation of the `smoothed_rtt` due to a misreporting peer.

Additionally, an endpoint might postpone the processing of acknowledgments when the corresponding decryption keys are not immediately available. For example, a client might receive an acknowledgment for a 0-RTT packet that it cannot decrypt because 1-RTT packet protection keys are not yet available to it. In such cases, an endpoint SHOULD subtract such local delays from its RTT sample until the handshake is confirmed.

Similar to [RFC6298], `smoothed_rtt` and `rttvar` are computed as follows.

An endpoint initializes the RTT estimator during connection establishment and when the estimator is reset during connection migration; see Section 9.4 of [QUIC-TRANSPORT]. Before any RTT samples are available for a new path or when the estimator is reset, the estimator is initialized using the initial RTT; see Section 6.2.2.

`smoothed_rtt` and `rttvar` are initialized as follows, where `kInitialRtt` contains the initial RTT value:

```
smoothed_rtt = kInitialRtt
rttvar = kInitialRtt / 2
```

RTT samples for the network path are recorded in `latest_rtt`; see Section 5.1. On the first RTT sample after initialization, the estimator is reset using that sample. This ensures that the estimator retains no history of past samples.

On the first RTT sample after initialization, `smoothed_rtt` and `rttvar` are set as follows:


```
smoothed_rtt = latest_rtt
rttvar = latest_rtt / 2
```

On subsequent RTT samples, `smoothed_rtt` and `rttvar` evolve as follows:

```
ack_delay = decoded acknowledgment delay from ACK frame
if (handshake confirmed):
    ack_delay = min(ack_delay, max_ack_delay)
adjusted_rtt = latest_rtt
if (min_rtt + ack_delay < latest_rtt):
    adjusted_rtt = latest_rtt - ack_delay
smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * adjusted_rtt
rttvar_sample = abs(smoothed_rtt - adjusted_rtt)
rttvar = 3/4 * rttvar + 1/4 * rttvar_sample
```

6. Loss Detection

QUIC senders use acknowledgments to detect lost packets, and a probe time out (see Section 6.2) to ensure acknowledgments are received. This section provides a description of these algorithms.

If a packet is lost, the QUIC transport needs to recover from that loss, such as by retransmitting the data, sending an updated frame, or discarding the frame. For more information, see Section 13.3 of [QUIC-TRANSPORT].

Loss detection is separate per packet number space, unlike RTT measurement and congestion control, because RTT and congestion control are properties of the path, whereas loss detection also relies upon key availability.

6.1. Acknowledgment-Based Detection

Acknowledgment-based loss detection implements the spirit of TCP's Fast Retransmit ([RFC5681]), Early Retransmit ([RFC5827]), FACK ([FACK]), SACK loss recovery ([RFC6675]), and RACK-TLP ([RACK]). This section provides an overview of how these algorithms are implemented in QUIC.

A packet is declared lost if it meets all the following conditions:

- * The packet is unacknowledged, in-flight, and was sent prior to an acknowledged packet.
- * The packet was sent `kPacketThreshold` packets before an acknowledged packet (Section 6.1.1), or it was sent long enough in the past (Section 6.1.2).

The acknowledgment indicates that a packet sent later was delivered, and the packet and time thresholds provide some tolerance for packet reordering.

Spuriously declaring packets as lost leads to unnecessary retransmissions and may result in degraded performance due to the actions of the congestion controller upon detecting loss. Implementations can detect spurious retransmissions and increase the reordering threshold in packets or time to reduce future spurious retransmissions and loss events. Implementations with adaptive time thresholds MAY choose to start with smaller initial reordering thresholds to minimize recovery latency.

6.1.1. Packet Threshold

The RECOMMENDED initial value for the packet reordering threshold (`kPacketThreshold`) is 3, based on best practices for TCP loss detection ([RFC5681], [RFC6675]). In order to remain similar to TCP, implementations SHOULD NOT use a packet threshold less than 3; see [RFC5681].

Some networks may exhibit higher degrees of packet reordering, causing a sender to detect spurious losses. Additionally, packet reordering could be more common with QUIC than TCP, because network elements that could observe and reorder TCP packets cannot do that for QUIC, because QUIC packet numbers are encrypted. Algorithms that increase the reordering threshold after spuriously detecting losses, such as RACK [RACK], have proven to be useful in TCP and are expected to be at least as useful in QUIC.

6.1.2. Time Threshold

Once a later packet within the same packet number space has been acknowledged, an endpoint SHOULD declare an earlier packet lost if it was sent a threshold amount of time in the past. To avoid declaring packets as lost too early, this time threshold MUST be set to at least the local timer granularity, as indicated by the `kGranularity` constant. The time threshold is:

$$\max(kTimeThreshold * \max(smoothed_rtt, latest_rtt), kGranularity)$$

If packets sent prior to the largest acknowledged packet cannot yet be declared lost, then a timer SHOULD be set for the remaining time.

Using $\max(smoothed_rtt, latest_rtt)$ protects from the two following cases:

- * the latest RTT sample is lower than the smoothed RTT, perhaps due to reordering where the acknowledgment encountered a shorter path;
- * the latest RTT sample is higher than the smoothed RTT, perhaps due to a sustained increase in the actual RTT, but the smoothed RTT has not yet caught up.

The RECOMMENDED time threshold (`kTimeThreshold`), expressed as a round-trip time multiplier, is $9/8$. The RECOMMENDED value of the timer granularity (`kGranularity`) is 1ms.

Note: TCP's RACK ([RACK]) specifies a slightly larger threshold, equivalent to $5/4$, for a similar purpose. Experience with QUIC shows that $9/8$ works well.

Implementations MAY experiment with absolute thresholds, thresholds from previous connections, adaptive thresholds, or including RTT variation. Smaller thresholds reduce reordering resilience and increase spurious retransmissions, and larger thresholds increase loss detection delay.

6.2. Probe Timeout

A Probe Timeout (PTO) triggers sending one or two probe datagrams when ack-eliciting packets are not acknowledged within the expected period of time or the server may not have validated the client's address. A PTO enables a connection to recover from loss of tail packets or acknowledgments.

As with loss detection, the probe timeout is per packet number space. That is, a PTO value is computed per packet number space.

A PTO timer expiration event does not indicate packet loss and MUST NOT cause prior unacknowledged packets to be marked as lost. When an acknowledgment is received that newly acknowledges packets, loss detection proceeds as dictated by packet and time threshold mechanisms; see Section 6.1.

The PTO algorithm used in QUIC implements the reliability functions of Tail Loss Probe [RACK], RTO [RFC5681], and F-RTO algorithms for TCP [RFC5682]. The timeout computation is based on TCP's retransmission timeout period [RFC6298].

6.2.1. Computing PTO

When an ack-eliciting packet is transmitted, the sender schedules a timer for the PTO period as follows:

$$PTO = smoothed_rtt + \max(4 * rttvar, kGranularity) + max_ack_delay$$

The PTO period is the amount of time that a sender ought to wait for an acknowledgment of a sent packet. This time period includes the estimated network roundtrip-time (`smoothed_rtt`), the variation in the estimate ($4 * rttvar$), and `max_ack_delay`, to account for the maximum time by which a receiver might delay sending an acknowledgment.

When the PTO is armed for Initial or Handshake packet number spaces, the `max_ack_delay` in the PTO period computation is set to 0, since the peer is expected to not delay these packets intentionally; see 13.2.1 of [QUIC-TRANSPORT].

The PTO period MUST be at least `kGranularity`, to avoid the timer expiring immediately.

When ack-eliciting packets in multiple packet number spaces are in flight, the timer MUST be set to the earlier value of the Initial and Handshake packet number spaces.

An endpoint MUST NOT set its PTO timer for the application data packet number space until the handshake is confirmed. Doing so prevents the endpoint from retransmitting information in packets when either the peer does not yet have the keys to process them or the endpoint does not yet have the keys to process their acknowledgments. For example, this can happen when a client sends 0-RTT packets to the server; it does so without knowing whether the server will be able to decrypt them. Similarly, this can happen when a server sends 1-RTT packets before confirming that the client has verified the server's certificate and can therefore read these 1-RTT packets.

A sender SHOULD restart its PTO timer every time an ack-eliciting packet is sent or acknowledged, or when Initial or Handshake keys are discarded (Section 4.9 of [QUIC-TLS]). This ensures the PTO is always set based on the latest estimate of the round-trip time and for the correct packet across packet number spaces.

When a PTO timer expires, the PTO backoff MUST be increased, resulting in the PTO period being set to twice its current value. The PTO backoff factor is reset when an acknowledgment is received, except in the following case. A server might take longer to respond to packets during the handshake than otherwise. To protect such a server from repeated client probes, the PTO backoff is not reset at a client that is not yet certain that the server has finished validating the client's address. That is, a client does not reset the PTO backoff factor on receiving acknowledgments in Initial packets.

This exponential reduction in the sender's rate is important because consecutive PTOs might be caused by loss of packets or acknowledgments due to severe congestion. Even when there are acknowledging packets in-flight in multiple packet number spaces, the exponential increase in probe timeout occurs across all spaces to prevent excess load on the network. For example, a timeout in the Initial packet number space doubles the length of the timeout in the Handshake packet number space.

The total length of time over which consecutive PTOs expire is limited by the idle timeout.

The PTO timer MUST NOT be set if a timer is set for time threshold loss detection; see Section 6.1.2. A timer that is set for time threshold loss detection will expire earlier than the PTO timer in most cases and is less likely to spuriously retransmit data.

6.2.2. Handshakes and New Paths

Resumed connections over the same network MAY use the previous connection's final smoothed RTT value as the resumed connection's initial RTT. When no previous RTT is available, the initial RTT SHOULD be set to 333ms. This results in handshakes starting with a PTO of 1 second, as recommended for TCP's initial retransmission timeout; see Section 2 of [RFC6298].

A connection MAY use the delay between sending a PATH_CHALLENGE and receiving a PATH_RESPONSE to set the initial RTT (see `kInitialRtt` in Appendix A.2) for a new path, but the delay SHOULD NOT be considered an RTT sample.

Initial packets and Handshake packets could be never acknowledged, but they are removed from bytes in flight when the Initial and Handshake keys are discarded, as described below in Section 6.4. When Initial or Handshake keys are discarded, the PTO and loss detection timers MUST be reset, because discarding keys indicates forward progress and the loss detection timer might have been set for a now discarded packet number space.

6.2.2.1. Before Address Validation

Until the server has validated the client's address on the path, the amount of data it can send is limited to three times the amount of data received, as specified in Section 8.1 of [QUIC-TRANSPORT]. If no additional data can be sent, the server's PTO timer MUST NOT be armed until datagrams have been received from the client, because packets sent on PTO count against the anti-amplification limit. Note that the server could fail to validate the client's address even if 0-RTT is accepted.

Since the server could be blocked until more datagrams are received from the client, it is the client's responsibility to send packets to unblock the server until it is certain that the server has finished its address validation (see Section 8 of [QUIC-TRANSPORT]). That is, the client MUST set the probe timer if the client has not received an acknowledgment for any of its Handshake packets and the handshake is not confirmed (see Section 4.1.2 of [QUIC-TLS]), even if there are no packets in flight. When the PTO fires, the client MUST send a Handshake packet if it has Handshake keys, otherwise it MUST send an Initial packet in a UDP datagram with a payload of at least 1200 bytes.

6.2.3. Speeding Up Handshake Completion

When a server receives an Initial packet containing duplicate CRYPTO data, it can assume the client did not receive all of the server's CRYPTO data sent in Initial packets, or the client's estimated RTT is too small. When a client receives Handshake or 1-RTT packets prior to obtaining Handshake keys, it may assume some or all of the server's Initial packets were lost.

To speed up handshake completion under these conditions, an endpoint MAY, for a limited number of times per connection, send a packet containing unacknowledged CRYPTO data earlier than the PTO expiry, subject to the address validation limits in Section 8.1 of [QUIC-TRANSPORT]. Doing so at most once for each connection is adequate to quickly recover from a single packet loss. An endpoint that always retransmits packets in response to receiving packets that it cannot process risks creating an infinite exchange of packets.

Endpoints can also use coalesced packets (see Section 12.2 of [QUIC-TRANSPORT]) to ensure that each datagram elicits at least one acknowledgment. For example, a client can coalesce an Initial packet containing PING and PADDING frames with a 0-RTT data packet and a server can coalesce an Initial packet containing a PING frame with one or more packets in its first flight.

6.2.4. Sending Probe Packets

When a PTO timer expires, a sender **MUST** send at least one ack-eliciting packet in the packet number space as a probe. An endpoint **MAY** send up to two full-sized datagrams containing ack-eliciting packets, to avoid an expensive consecutive PTO expiration due to a single lost datagram, or transmit data from multiple packet number spaces. All probe packets sent on a PTO **MUST** be ack-eliciting.

In addition to sending data in the packet number space for which the timer expired, the sender **SHOULD** send ack-eliciting packets from other packet number spaces with in-flight data, coalescing packets if possible. This is particularly valuable when the server has both Initial and Handshake data in-flight or the client has both Handshake and Application Data in-flight, because the peer might only have receive keys for one of the two packet number spaces.

If the sender wants to elicit a faster acknowledgment on PTO, it can skip a packet number to eliminate the acknowledgment delay.

An endpoint **SHOULD** include new data in packets that are sent on PTO expiration. Previously sent data **MAY** be sent if no new data can be sent. Implementations **MAY** use alternative strategies for determining the content of probe packets, including sending new or retransmitted data based on the application's priorities.

It is possible the sender has no new or previously-sent data to send. As an example, consider the following sequence of events: new application data is sent in a STREAM frame, deemed lost, then retransmitted in a new packet, and then the original transmission is acknowledged. When there is no data to send, the sender **SHOULD** send a PING or other ack-eliciting frame in a single packet, re-arming the PTO timer.

Alternatively, instead of sending an ack-eliciting packet, the sender **MAY** mark any packets still in flight as lost. Doing so avoids sending an additional packet, but increases the risk that loss is declared too aggressively, resulting in an unnecessary rate reduction by the congestion controller.

Consecutive PTO periods increase exponentially, and as a result, connection recovery latency increases exponentially as packets continue to be dropped in the network. Sending two packets on PTO expiration increases resilience to packet drops, thus reducing the probability of consecutive PTO events.

When the PTO timer expires multiple times and new data cannot be sent, implementations must choose between sending the same payload every time or sending different payloads. Sending the same payload may be simpler and ensures the highest priority frames arrive first. Sending different payloads each time reduces the chances of spurious retransmission.

6.3. Handling Retry Packets

A Retry packet causes a client to send another Initial packet, effectively restarting the connection process. A Retry packet indicates that the Initial was received, but not processed. A Retry packet cannot be treated as an acknowledgment, because it does not indicate that a packet was processed or specify the packet number.

Clients that receive a Retry packet reset congestion control and loss recovery state, including resetting any pending timers. Other connection state, in particular cryptographic handshake messages, is retained; see Section 17.2.5 of [QUIC-TRANSPORT].

The client MAY compute an RTT estimate to the server as the time period from when the first Initial was sent to when a Retry or a Version Negotiation packet is received. The client MAY use this value in place of its default for the initial RTT estimate.

6.4. Discarding Keys and Packet State

When Initial and Handshake packet protection keys are discarded (see Section 4.9 of [QUIC-TLS]), all packets that were sent with those keys can no longer be acknowledged because their acknowledgments cannot be processed. The sender MUST discard all recovery state associated with those packets and MUST remove them from the count of bytes in flight.

Endpoints stop sending and receiving Initial packets once they start exchanging Handshake packets; see Section 17.2.2.1 of [QUIC-TRANSPORT]. At this point, recovery state for all in-flight Initial packets is discarded.

When 0-RTT is rejected, recovery state for all in-flight 0-RTT packets is discarded.

If a server accepts 0-RTT, but does not buffer 0-RTT packets that arrive before Initial packets, early 0-RTT packets will be declared lost, but that is expected to be infrequent.

It is expected that keys are discarded after packets encrypted with them would be acknowledged or declared lost. However, Initial and Handshake secrets are discarded as soon as handshake and 1-RTT keys are proven to be available to both client and server; see Section 4.9.1 of [QUIC-TLS].

7. Congestion Control

This document specifies a sender-side congestion controller for QUIC similar to TCP NewReno ([RFC6582]).

The signals QUIC provides for congestion control are generic and are designed to support different sender-side algorithms. A sender can unilaterally choose a different algorithm to use, such as Cubic ([RFC8312]).

If a sender uses a different controller than that specified in this document, the chosen controller MUST conform to the congestion control guidelines specified in Section 3.1 of [RFC8085].

Similar to TCP, packets containing only ACK frames do not count towards bytes in flight and are not congestion controlled. Unlike TCP, QUIC can detect the loss of these packets and MAY use that information to adjust the congestion controller or the rate of ACK-only packets being sent, but this document does not describe a mechanism for doing so.

The algorithm in this document specifies and uses the controller's congestion window in bytes.

An endpoint MUST NOT send a packet if it would cause `bytes_in_flight` (see Appendix B.2) to be larger than the congestion window, unless the packet is sent on a PTO timer expiration (see Section 6.2) or when entering recovery (see Section 7.3.2).

7.1. Explicit Congestion Notification

If a path has been validated to support ECN ([RFC3168], [RFC8311]), QUIC treats a Congestion Experienced (CE) codepoint in the IP header as a signal of congestion. This document specifies an endpoint's response when the peer-reported ECN-CE count increases; see Section 13.4.2 of [QUIC-TRANSPORT].

7.2. Initial and Minimum Congestion Window

QUIC begins every connection in slow start with the congestion window set to an initial value. Endpoints SHOULD use an initial congestion window of 10 times the maximum datagram size (`max_datagram_size`), while limiting the window to the larger of 14720 bytes or twice the maximum datagram size. This follows the analysis and recommendations in [RFC6928], increasing the byte limit to account for the smaller 8-byte overhead of UDP compared to the 20-byte overhead for TCP.

If the maximum datagram size changes during the connection, the initial congestion window SHOULD be recalculated with the new size. If the maximum datagram size is decreased in order to complete the handshake, the congestion window SHOULD be set to the new initial congestion window.

Prior to validating the client's address, the server can be further limited by the anti-amplification limit as specified in Section 8.1 of [QUIC-TRANSPORT]. Though the anti-amplification limit can prevent the congestion window from being fully utilized and therefore slow down the increase in congestion window, it does not directly affect the congestion window.

The minimum congestion window is the smallest value the congestion window can decrease to as a response to loss, increase in the peer-reported ECN-CE count, or persistent congestion. The RECOMMENDED value is $2 * \text{max_datagram_size}$.

7.3. Congestion Control States

The NewReno congestion controller described in this document has three distinct states, as shown in Figure 1.

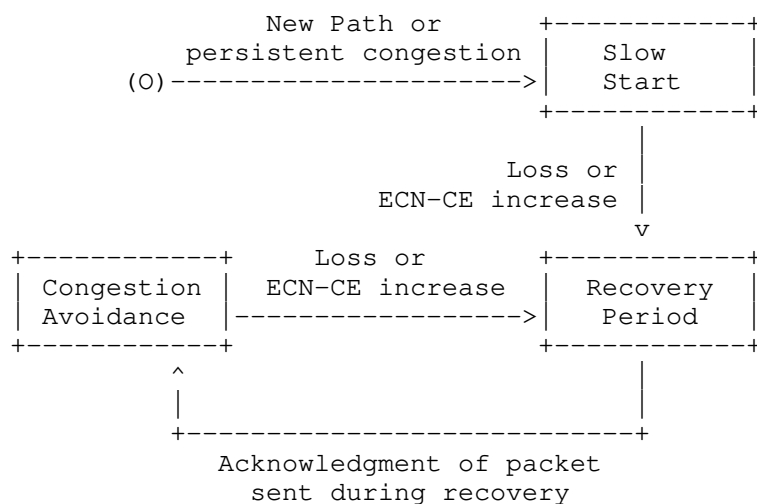


Figure 1: Congestion Control States and Transitions

These states and the transitions between them are described in subsequent sections.

7.3.1. Slow Start

A NewReno sender is in slow start any time the congestion window is below the slow start threshold. A sender begins in slow start because the slow start threshold is initialized to an infinite value.

While a sender is in slow start, the congestion window increases by the number of bytes acknowledged when each acknowledgment is processed. This results in exponential growth of the congestion window.

The sender MUST exit slow start and enter a recovery period when a packet is lost or when the ECN-CE count reported by its peer increases.

A sender re-enters slow start any time the congestion window is less than the slow start threshold, which only occurs after persistent congestion is declared.

7.3.2. Recovery

A NewReno sender enters a recovery period when it detects the loss of a packet or the ECN-CE count reported by its peer increases. A sender that is already in a recovery period stays in it and does not re-enter it.

On entering a recovery period, a sender MUST set the slow start threshold to half the value of the congestion window when loss is detected. The congestion window MUST be set to the reduced value of the slow start threshold before exiting the recovery period.

Implementations MAY reduce the congestion window immediately upon entering a recovery period or use other mechanisms, such as Proportional Rate Reduction ([PRR]), to reduce the congestion window more gradually. If the congestion window is reduced immediately, a single packet can be sent prior to reduction. This speeds up loss recovery if the data in the lost packet is retransmitted and is similar to TCP as described in Section 5 of [RFC6675].

The recovery period aims to limit congestion window reduction to once per round trip. Therefore during a recovery period, the congestion window does not change in response to new losses or increases in the ECN-CE count.

A recovery period ends and the sender enters congestion avoidance when a packet sent during the recovery period is acknowledged. This is slightly different from TCP's definition of recovery, which ends when the lost segment that started recovery is acknowledged ([RFC5681]).

7.3.3. Congestion Avoidance

A NewReno sender is in congestion avoidance any time the congestion window is at or above the slow start threshold and not in a recovery period.

A sender in congestion avoidance uses an Additive Increase Multiplicative Decrease (AIMD) approach that MUST limit the increase to the congestion window to at most one maximum datagram size for each congestion window that is acknowledged.

The sender exits congestion avoidance and enters a recovery period when a packet is lost or when the ECN-CE count reported by its peer increases.

7.4. Ignoring Loss of Undecryptable Packets

During the handshake, some packet protection keys might not be available when a packet arrives and the receiver can choose to drop the packet. In particular, Handshake and 0-RTT packets cannot be processed until the Initial packets arrive and 1-RTT packets cannot be processed until the handshake completes. Endpoints MAY ignore the loss of Handshake, 0-RTT, and 1-RTT packets that might have arrived before the peer had packet protection keys to process those packets.

Endpoints MUST NOT ignore the loss of packets that were sent after the earliest acknowledged packet in a given packet number space.

7.5. Probe Timeout

Probe packets MUST NOT be blocked by the congestion controller. A sender MUST however count these packets as being additionally in flight, since these packets add network load without establishing packet loss. Note that sending probe packets might cause the sender's bytes in flight to exceed the congestion window until an acknowledgment is received that establishes loss or delivery of packets.

7.6. Persistent Congestion

When a sender establishes loss of all packets sent over a long enough duration, the network is considered to be experiencing persistent congestion.

7.6.1. Duration

The persistent congestion duration is computed as follows:

$$(\text{smoothed_rtt} + \max(4 * \text{rttvar}, k\text{Granularity}) + \text{max_ack_delay}) * k\text{PersistentCongestionThreshold}$$

Unlike the PTO computation in Section 6.2, this duration includes the `max_ack_delay` irrespective of the packet number spaces in which losses are established.

This duration allows a sender to send as many packets before establishing persistent congestion, including some in response to PTO expiration, as TCP does with Tail Loss Probes ([RACK]) and a Retransmission Timeout ([RFC5681]).

Larger values of `kPersistentCongestionThreshold` cause the sender to become less responsive to persistent congestion in the network, which can result in aggressive sending into a congested network. Too small a value can result in a sender declaring persistent congestion unnecessarily, resulting in reduced throughput for the sender.

The RECOMMENDED value for `kPersistentCongestionThreshold` is 3, which results in behavior that is approximately equivalent to a TCP sender declaring an RTO after two TLPs.

This design does not use consecutive PTO events to establish persistent congestion, since application patterns impact PTO expirations. For example, a sender that sends small amounts of data

with silence periods between them restarts the PTO timer every time it sends, potentially preventing the PTO timer from expiring for a long period of time, even when no acknowledgments are being received. The use of a duration enables a sender to establish persistent congestion without depending on PTO expiration.

7.6.2. Establishing Persistent Congestion

A sender establishes persistent congestion after the receipt of an acknowledgment if two packets that are ack-eliciting are declared lost, and:

- * across all packet number spaces, none of the packets sent between the send times of these two packets are acknowledged;
- * the duration between the send times of these two packets exceeds the persistent congestion duration (Section 7.6.1); and
- * a prior RTT sample existed when these two packets were sent.

These two packets MUST be ack-eliciting, since a receiver is required to acknowledge only ack-eliciting packets within its maximum ack delay; see Section 13.2 of [QUIC-TRANSPORT].

The persistent congestion period SHOULD NOT start until there is at least one RTT sample. Before the first RTT sample, a sender arms its PTO timer based on the initial RTT (Section 6.2.2), which could be substantially larger than the actual RTT. Requiring a prior RTT sample prevents a sender from establishing persistent congestion with potentially too few probes.

Since network congestion is not affected by packet number spaces, persistent congestion SHOULD consider packets sent across packet number spaces. A sender that does not have state for all packet number spaces or an implementation that cannot compare send times across packet number spaces MAY use state for just the packet number space that was acknowledged. This might result in erroneously declaring persistent congestion, but it will not lead to a failure to detect persistent congestion.

When persistent congestion is declared, the sender's congestion window MUST be reduced to the minimum congestion window (`kMinimumWindow`), similar to a TCP sender's response on an RTO ([RFC5681]).

7.6.3. Example

The following example illustrates how a sender might establish persistent congestion. Assume:

$\text{smoothed_rtt} + \max(4 * \text{rttvar}, \text{kGranularity}) + \text{max_ack_delay} = 2$
 $\text{kPersistentCongestionThreshold} = 3$

Consider the following sequence of events:

Time	Action
t=0	Send packet #1 (app data)
t=1	Send packet #2 (app data)
t=1.2	Recv acknowledgment of #1
t=2	Send packet #3 (app data)
t=3	Send packet #4 (app data)
t=4	Send packet #5 (app data)
t=5	Send packet #6 (app data)
t=6	Send packet #7 (app data)
t=8	Send packet #8 (PTO 1)
t=12	Send packet #9 (PTO 2)
t=12.2	Recv acknowledgment of #9

Table 1

Packets 2 through 8 are declared lost when the acknowledgment for packet 9 is received at $t = 12.2$.

The congestion period is calculated as the time between the oldest and newest lost packets: $8 - 1 = 7$. The persistent congestion duration is: $2 * 3 = 6$. Because the threshold was reached and because none of the packets between the oldest and the newest lost packets were acknowledged, the network is considered to have experienced persistent congestion.

While this example shows PTO expiration, they are not required for persistent congestion to be established.

7.7. Pacing

A sender SHOULD pace sending of all in-flight packets based on input from the congestion controller.

Sending multiple packets into the network without any delay between them creates a packet burst that might cause short-term congestion and losses. Senders MUST either use pacing or limit such bursts. Senders SHOULD limit bursts to the initial congestion window; see Section 7.2. A sender with knowledge that the network path to the receiver can absorb larger bursts MAY use a higher limit.

An implementation should take care to architect its congestion controller to work well with a pacer. For instance, a pacer might wrap the congestion controller and control the availability of the congestion window, or a pacer might pace out packets handed to it by the congestion controller.

Timely delivery of ACK frames is important for efficient loss recovery. Packets containing only ACK frames SHOULD therefore not be paced, to avoid delaying their delivery to the peer.

Endpoints can implement pacing as they choose. A perfectly paced sender spreads packets exactly evenly over time. For a window-based congestion controller, such as the one in this document, that rate can be computed by averaging the congestion window over the round-trip time. Expressed as a rate in units of bytes per time, where `congestion_window` is in bytes:

$$\text{rate} = N * \text{congestion_window} / \text{smoothed_rtt}$$

Or, expressed as an inter-packet interval in units of time:

$$\text{interval} = (\text{smoothed_rtt} * \text{packet_size} / \text{congestion_window}) / N$$

Using a value for "N" that is small, but at least 1 (for example, 1.25) ensures that variations in round-trip time do not result in under-utilization of the congestion window.

Practical considerations, such as packetization, scheduling delays, and computational efficiency, can cause a sender to deviate from this rate over time periods that are much shorter than a round-trip time.

One possible implementation strategy for pacing uses a leaky bucket algorithm, where the capacity of the "bucket" is limited to the maximum burst size and the rate the "bucket" fills is determined by the above function.

7.8. Under-utilizing the Congestion Window

When bytes in flight is smaller than the congestion window and sending is not pacing limited, the congestion window is under-utilized. When this occurs, the congestion window SHOULD NOT be increased in either slow start or congestion avoidance. This can happen due to insufficient application data or flow control limits.

A sender that paces packets (see Section 7.7) might delay sending packets and not fully utilize the congestion window due to this delay. A sender SHOULD NOT consider itself application limited if it would have fully utilized the congestion window without pacing delay.

A sender MAY implement alternative mechanisms to update its congestion window after periods of under-utilization, such as those proposed for TCP in [RFC7661].

8. Security Considerations

8.1. Loss and Congestion Signals

Loss detection and congestion control fundamentally involve consumption of signals, such as delay, loss, and ECN markings, from unauthenticated entities. An attacker can cause endpoints to reduce their sending rate by manipulating these signals; by dropping packets, by altering path delay strategically, or by changing ECN codepoints.

8.2. Traffic Analysis

Packets that carry only ACK frames can be heuristically identified by observing packet size. Acknowledgment patterns may expose information about link characteristics or application behavior. To reduce leaked information, endpoints can bundle acknowledgments with other frames, or they can use PADDING frames at a potential cost to performance.

8.3. Misreporting ECN Markings

A receiver can misreport ECN markings to alter the congestion response of a sender. Suppressing reports of ECN-CE markings could cause a sender to increase their send rate. This increase could result in congestion and loss.

A sender can detect suppression of reports by marking occasional packets that it sends with an ECN-CE marking. If a packet sent with an ECN-CE marking is not reported as having been CE marked when the packet is acknowledged, then the sender can disable ECN for that path by not setting ECT codepoints in subsequent packets sent on that path [RFC3168].

Reporting additional ECN-CE markings will cause a sender to reduce their sending rate, which is similar in effect to advertising reduced connection flow control limits and so no advantage is gained by doing so.

Endpoints choose the congestion controller that they use. Congestion controllers respond to reports of ECN-CE by reducing their rate, but the response may vary. Markings can be treated as equivalent to loss ([RFC3168]), but other responses can be specified, such as ([RFC8511]) or ([RFC8311]).

9. IANA Considerations

This document has no IANA actions.

10. References

10.1. Normative References

[QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-34, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-tls-34>>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-transport-34>>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.

- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

10.2. Informative References

- [FACK] Mathis, M. and J. Mahdavi, "Forward Acknowledgement: Refining TCP Congestion Control", ACM SIGCOMM , August 1996.
- [PRR] Mathis, M., Dukkupati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", RFC 6937, DOI 10.17487/RFC6937, May 2013, <<https://www.rfc-editor.org/info/rfc6937>>.
- [RACK] Cheng, Y., Cardwell, N., Dukkupati, N., and P. Jha, "The RACK-TLP loss detection algorithm for TCP", Work in Progress, Internet-Draft, draft-ietf-tcpm-rack-15, 22 December 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-tcpm-rack-15.txt>>.
- [RETRANSMISSION] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", ACM SIGCOMM CCR , January 1995.
- [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <<https://www.rfc-editor.org/info/rfc2018>>.
- [RFC3465] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, DOI 10.17487/RFC3465, February 2003, <<https://www.rfc-editor.org/info/rfc3465>>.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <<https://www.rfc-editor.org/info/rfc5681>>.
- [RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, DOI 10.17487/RFC5682, September 2009, <<https://www.rfc-editor.org/info/rfc5682>>.

- [RFC5827] Allman, M., Avrachenkov, K., Ayesta, U., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, DOI 10.17487/RFC5827, May 2010, <<https://www.rfc-editor.org/info/rfc5827>>.
- [RFC6297] Welzl, M. and D. Ros, "A Survey of Lower-than-Best-Effort Transport Protocols", RFC 6297, DOI 10.17487/RFC6297, June 2011, <<https://www.rfc-editor.org/info/rfc6297>>.
- [RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <<https://www.rfc-editor.org/info/rfc6298>>.
- [RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <<https://www.rfc-editor.org/info/rfc6582>>.
- [RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, DOI 10.17487/RFC6675, August 2012, <<https://www.rfc-editor.org/info/rfc6675>>.
- [RFC6928] Chu, J., Dukkkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <<https://www.rfc-editor.org/info/rfc6928>>.
- [RFC7661] Fairhurst, G., Sathiaselalan, A., and R. Secchi, "Updating TCP to Support Rate-Limited Traffic", RFC 7661, DOI 10.17487/RFC7661, October 2015, <<https://www.rfc-editor.org/info/rfc7661>>.
- [RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <<https://www.rfc-editor.org/info/rfc8311>>.
- [RFC8312] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <<https://www.rfc-editor.org/info/rfc8312>>.

[RFC8511] Khademi, N., Welzl, M., Armitage, G., and G. Fairhurst, "TCP Alternative Backoff with ECN (ABE)", RFC 8511, DOI 10.17487/RFC8511, December 2018, <<https://www.rfc-editor.org/info/rfc8511>>.

Appendix A. Loss Recovery Pseudocode

We now describe an example implementation of the loss detection mechanisms described in Section 6.

The pseudocode segments in this section are licensed as Code Components; see the copyright notice.

A.1. Tracking Sent Packets

To correctly implement congestion control, a QUIC sender tracks every ack-eliciting packet until the packet is acknowledged or lost. It is expected that implementations will be able to access this information by packet number and crypto context and store the per-packet fields (Appendix A.1.1) for loss recovery and congestion control.

After a packet is declared lost, the endpoint can still maintain state for it for an amount of time to allow for packet reordering; see Section 13.3 of [QUIC-TRANSPORT]. This enables a sender to detect spurious retransmissions.

Sent packets are tracked for each packet number space, and ACK processing only applies to a single space.

A.1.1. Sent Packet Fields

packet_number: The packet number of the sent packet.

ack_eliciting: A boolean that indicates whether a packet is ack-eliciting. If true, it is expected that an acknowledgment will be received, though the peer could delay sending the ACK frame containing it by up to the `max_ack_delay`.

in_flight: A boolean that indicates whether the packet counts towards bytes in flight.

sent_bytes: The number of bytes sent in the packet, not including UDP or IP overhead, but including QUIC framing overhead.

time_sent: The time the packet was sent.

A.2. Constants of Interest

Constants used in loss recovery are based on a combination of RFCs, papers, and common practice.

kPacketThreshold: Maximum reordering in packets before packet threshold loss detection considers a packet lost. The value recommended in Section 6.1.1 is 3.

kTimeThreshold: Maximum reordering in time before time threshold loss detection considers a packet lost. Specified as an RTT multiplier. The value recommended in Section 6.1.2 is 9/8.

kGranularity: Timer granularity. This is a system-dependent value, and Section 6.1.2 recommends a value of 1ms.

kInitialRtt: The RTT used before an RTT sample is taken. The value recommended in Section 6.2.2 is 333ms.

kPacketNumberSpace: An enum to enumerate the three packet number spaces.

```
enum kPacketNumberSpace {  
    Initial,  
    Handshake,  
    ApplicationData,  
}
```

A.3. Variables of interest

Variables required to implement the congestion control mechanisms are described in this section.

latest_rtt: The most recent RTT measurement made when receiving an ack for a previously unacked packet.

smoothed_rtt: The smoothed RTT of the connection, computed as described in Section 5.3.

rttvar: The RTT variation, computed as described in Section 5.3.

min_rtt: The minimum RTT seen over a period of time, ignoring acknowledgment delay, as described in Section 5.2.

first_rtt_sample: The time that the first RTT sample was obtained.

max_ack_delay: The maximum amount of time by which the receiver

intends to delay acknowledgments for packets in the Application Data packet number space, as defined by the eponymous transport parameter (Section 18.2 of [QUIC-TRANSPORT]). Note that the actual `ack_delay` in a received ACK frame may be larger due to late timers, reordering, or loss.

`loss_detection_timer`: Multi-modal timer used for loss detection.

`pto_count`: The number of times a PTO has been sent without receiving an ack.

`time_of_last_ack_eliciting_packet[kPacketNumberSpace]`: The time the most recent ack-eliciting packet was sent.

`largest_acked_packet[kPacketNumberSpace]`: The largest packet number acknowledged in the packet number space so far.

`loss_time[kPacketNumberSpace]`: The time at which the next packet in that packet number space can be considered lost based on exceeding the reordering window in time.

`sent_packets[kPacketNumberSpace]`: An association of packet numbers in a packet number space to information about them. Described in detail above in Appendix A.1.

A.4. Initialization

At the beginning of the connection, initialize the loss detection variables as follows:

```
loss_detection_timer.reset()
pto_count = 0
latest_rtt = 0
smoothed_rtt = kInitialRtt
rttvar = kInitialRtt / 2
min_rtt = 0
first_rtt_sample = 0
for pn_space in [ Initial, Handshake, ApplicationData ]:
    largest_acked_packet[pn_space] = infinite
    time_of_last_ack_eliciting_packet[pn_space] = 0
    loss_time[pn_space] = 0
```

A.5. On Sending a Packet

After a packet is sent, information about the packet is stored. The parameters to `OnPacketSent` are described in detail above in Appendix A.1.1.

Pseudocode for OnPacketSent follows:

```
OnPacketSent(packet_number, pn_space, ack_eliciting,
             in_flight, sent_bytes):
    sent_packets[pn_space][packet_number].packet_number =
        packet_number
    sent_packets[pn_space][packet_number].time_sent = now()
    sent_packets[pn_space][packet_number].ack_eliciting =
        ack_eliciting
    sent_packets[pn_space][packet_number].in_flight = in_flight
    sent_packets[pn_space][packet_number].sent_bytes = sent_bytes
    if (in_flight):
        if (ack_eliciting):
            time_of_last_ack_eliciting_packet[pn_space] = now()
        OnPacketSentCC(sent_bytes)
        SetLossDetectionTimer()
```

A.6. On Receiving a Datagram

When a server is blocked by anti-amplification limits, receiving a datagram unblocks it, even if none of the packets in the datagram are successfully processed. In such a case, the PTO timer will need to be re-armed.

Pseudocode for OnDatagramReceived follows:

```
OnDatagramReceived(datagram):
    // If this datagram unblocks the server, arm the
    // PTO timer to avoid deadlock.
    if (server was at anti-amplification limit):
        SetLossDetectionTimer()
```

A.7. On Receiving an Acknowledgment

When an ACK frame is received, it may newly acknowledge any number of packets.

Pseudocode for OnAckReceived and UpdateRtt follow:

```
IncludesAckEliciting(packets):
    for packet in packets:
        if (packet.ack_eliciting):
            return true
    return false

OnAckReceived(ack, pn_space):
    if (largest_acked_packet[pn_space] == infinite):
        largest_acked_packet[pn_space] = ack.largest_acked
```



```
else:
    largest_acked_packet[pn_space] =
        max(largest_acked_packet[pn_space], ack.largest_acked)

// DetectAndRemoveAkedPackets finds packets that are newly
// acknowledged and removes them from sent_packets.
newly_acked_packets =
    DetectAndRemoveAkedPackets(ack, pn_space)
// Nothing to do if there are no newly acked packets.
if (newly_acked_packets.empty()):
    return

// Update the RTT if the largest acknowledged is newly acked
// and at least one ack-eliciting was newly acked.
if (newly_acked_packets.largest().packet_number ==
    ack.largest_acked &&
    IncludesAckEliciting(newly_acked_packets)):
    latest_rtt =
        now() - newly_acked_packets.largest().time_sent
    UpdateRtt(ack.ack_delay)

// Process ECN information if present.
if (ACK frame contains ECN information):
    ProcessECN(ack, pn_space)

lost_packets = DetectAndRemoveLostPackets(pn_space)
if (!lost_packets.empty()):
    OnPacketsLost(lost_packets)
OnPacketsAked(newly_acked_packets)

// Reset pto_count unless the client is unsure if
// the server has validated the client's address.
if (PeerCompletedAddressValidation()):
    pto_count = 0
SetLossDetectionTimer()

UpdateRtt(ack_delay):
    if (first_rtt_sample == 0):
        min_rtt = latest_rtt
        smoothed_rtt = latest_rtt
        rttvar = latest_rtt / 2
        first_rtt_sample = now()
        return

// min_rtt ignores acknowledgment delay.
min_rtt = min(min_rtt, latest_rtt)
// Limit ack_delay by max_ack_delay after handshake
```

```

// confirmation.
if (handshake confirmed):
    ack_delay = min(ack_delay, max_ack_delay)

// Adjust for acknowledgment delay if plausible.
adjusted_rtt = latest_rtt
if (latest_rtt > min_rtt + ack_delay):
    adjusted_rtt = latest_rtt - ack_delay

rttvar = 3/4 * rttvar + 1/4 * abs(smoothed_rtt - adjusted_rtt)
smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * adjusted_rtt

```

A.8. Setting the Loss Detection Timer

QUIC loss detection uses a single timer for all timeout loss detection. The duration of the timer is based on the timer's mode, which is set in the packet and timer events further below. The function `SetLossDetectionTimer` defined below shows how the single timer is set.

This algorithm may result in the timer being set in the past, particularly if timers wake up late. Timers set in the past fire immediately.

Pseudocode for `SetLossDetectionTimer` follows (where the " $^$ " operator represents exponentiation):

```

GetLossTimeAndSpace():
    time = loss_time[Initial]
    space = Initial
    for pn_space in [ Handshake, ApplicationData ]:
        if (time == 0 || loss_time[pn_space] < time):
            time = loss_time[pn_space];
            space = pn_space
    return time, space

GetPtoTimeAndSpace():
    duration = (smoothed_rtt + max(4 * rttvar, kGranularity))
               * (2 ^ pto_count)
    // Arm PTO from now when there are no inflight packets.
    if (no in-flight packets):
        assert(!PeerCompletedAddressValidation())
        if (has handshake keys):
            return (now() + duration), Handshake
        else:
            return (now() + duration), Initial
    pto_timeout = infinite
    pto_space = Initial

```

```
for space in [ Initial, Handshake, ApplicationData ]:
    if (no in-flight packets in space):
        continue;
    if (space == ApplicationData):
        // Skip Application Data until handshake confirmed.
        if (handshake is not confirmed):
            return pto_timeout, pto_space
        // Include max_ack_delay and backoff for Application Data.
        duration += max_ack_delay * (2 ^ pto_count)

    t = time_of_last_ack_eliciting_packet[space] + duration
    if (t < pto_timeout):
        pto_timeout = t
        pto_space = space
    return pto_timeout, pto_space

PeerCompletedAddressValidation():
    // Assume clients validate the server's address implicitly.
    if (endpoint is server):
        return true
    // Servers complete address validation when a
    // protected packet is received.
    return has received Handshake ACK ||
        handshake confirmed

SetLossDetectionTimer():
    earliest_loss_time, _ = GetLossTimeAndSpace()
    if (earliest_loss_time != 0):
        // Time threshold loss detection.
        loss_detection_timer.update(earliest_loss_time)
        return

    if (server is at anti-amplification limit):
        // The server's timer is not set if nothing can be sent.
        loss_detection_timer.cancel()
        return

    if (no ack-eliciting packets in flight &&
        PeerCompletedAddressValidation()):
        // There is nothing to detect lost, so no timer is set.
        // However, the client needs to arm the timer if the
        // server might be blocked by the anti-amplification limit.
        loss_detection_timer.cancel()
        return

    timeout, _ = GetPtoTimeAndSpace()
    loss_detection_timer.update(timeout)
```

A.9. On Timeout

When the loss detection timer expires, the timer's mode determines the action to be performed.

Pseudocode for OnLossDetectionTimeout follows:

```
OnLossDetectionTimeout():
    earliest_loss_time, pn_space = GetLossTimeAndSpace()
    if (earliest_loss_time != 0):
        // Time threshold loss Detection
        lost_packets = DetectAndRemoveLostPackets(pn_space)
        assert(!lost_packets.empty())
        OnPacketsLost(lost_packets)
        SetLossDetectionTimer()
        return

    if (bytes_in_flight > 0):
        // PTO. Send new data if available, else retransmit old data.
        // If neither is available, send a single PING frame.
        _, pn_space = GetPtoTimeAndSpace()
        SendOneOrTwoAckElicitingPackets(pn_space)
    else:
        assert(!PeerCompletedAddressValidation())
        // Client sends an anti-deadlock packet: Initial is padded
        // to earn more anti-amplification credit,
        // a Handshake packet proves address ownership.
        if (has Handshake keys):
            SendOneAckElicitingHandshakePacket()
        else:
            SendOneAckElicitingPaddedInitialPacket()

    pto_count++
    SetLossDetectionTimer()
```

A.10. Detecting Lost Packets

DetectAndRemoveLostPackets is called every time an ACK is received or the time threshold loss detection timer expires. This function operates on the sent_packets for that packet number space and returns a list of packets newly detected as lost.

Pseudocode for DetectAndRemoveLostPackets follows:

```

DetectAndRemoveLostPackets(pn_space):
    assert(largest_acked_packet[pn_space] != infinite)
    loss_time[pn_space] = 0
    lost_packets = []
    loss_delay = kTimeThreshold * max(latest_rtt, smoothed_rtt)

    // Minimum time of kGranularity before packets are deemed lost.
    loss_delay = max(loss_delay, kGranularity)

    // Packets sent before this time are deemed lost.
    lost_send_time = now() - loss_delay

    foreach unacked in sent_packets[pn_space]:
        if (unacked.packet_number > largest_acked_packet[pn_space]):
            continue

        // Mark packet as lost, or set time when it should be marked.
        // Note: The use of kPacketThreshold here assumes that there
        // were no sender-induced gaps in the packet number space.
        if (unacked.time_sent <= lost_send_time ||
            largest_acked_packet[pn_space] >=
                unacked.packet_number + kPacketThreshold):
            sent_packets[pn_space].remove(unacked.packet_number)
            lost_packets.insert(unacked)
        else:
            if (loss_time[pn_space] == 0):
                loss_time[pn_space] = unacked.time_sent + loss_delay
            else:
                loss_time[pn_space] = min(loss_time[pn_space],
                                           unacked.time_sent + loss_delay)

    return lost_packets

```

A.11. Upon Dropping Initial or Handshake Keys

When Initial or Handshake keys are discarded, packets from the space are discarded and loss detection state is updated.

Pseudocode for OnPacketNumberSpaceDiscarded follows:

```

OnPacketNumberSpaceDiscarded(pn_space):
    assert(pn_space != ApplicationData)
    RemoveFromBytesInFlight(sent_packets[pn_space])
    sent_packets[pn_space].clear()
    // Reset the loss detection and PTO timer
    time_of_last_ack_eliciting_packet[pn_space] = 0
    loss_time[pn_space] = 0
    pto_count = 0
    SetLossDetectionTimer()

```

Appendix B. Congestion Control Pseudocode

We now describe an example implementation of the congestion controller described in Section 7.

The pseudocode segments in this section are licensed as Code Components; see the copyright notice.

B.1. Constants of interest

Constants used in congestion control are based on a combination of RFCs, papers, and common practice.

`kInitialWindow`: Default limit on the initial bytes in flight as described in Section 7.2.

`kMinimumWindow`: Minimum congestion window in bytes as described in Section 7.2.

`kLossReductionFactor`: Scaling factor applied to reduce the congestion window when a new loss event is detected. Section 7 recommends a value is 0.5.

`kPersistentCongestionThreshold`: Period of time for persistent congestion to be established, specified as a PTO multiplier. Section 7.6 recommends a value of 3.

B.2. Variables of interest

Variables required to implement the congestion control mechanisms are described in this section.

`max_datagram_size`: The sender's current maximum payload size. Does not include UDP or IP overhead. The max datagram size is used for congestion window computations. An endpoint sets the value of this variable based on its Path Maximum Transmission Unit (PMTU; see Section 14.2 of [QUIC-TRANSPORT]), with a minimum value of 1200 bytes.

`ecn_ce_counters[kPacketNumberSpace]`: The highest value reported for the ECN-CE counter in the packet number space by the peer in an ACK frame. This value is used to detect increases in the reported ECN-CE counter.

`bytes_in_flight`: The sum of the size in bytes of all sent packets that contain at least one ack-eliciting or PADDING frame, and have not been acknowledged or declared lost. The size does not include IP or UDP overhead, but does include the QUIC header and AEAD

overhead. Packets only containing ACK frames do not count towards bytes_in_flight to ensure congestion control does not impede congestion feedback.

congestion_window: Maximum number of bytes allowed to be in flight.

congestion_recovery_start_time: The time the current recovery period started due to the detection of loss or ECN. When a packet sent after this time is acknowledged, QUIC exits congestion recovery.

ssthresh: Slow start threshold in bytes. When the congestion window is below ssthresh, the mode is slow start and the window grows by the number of bytes acknowledged.

The congestion control pseudocode also accesses some of the variables from the loss recovery pseudocode.

B.3. Initialization

At the beginning of the connection, initialize the congestion control variables as follows:

```
congestion_window = kInitialWindow
bytes_in_flight = 0
congestion_recovery_start_time = 0
ssthresh = infinite
for pn_space in [ Initial, Handshake, ApplicationData ]:
    ecn_ce_counters[pn_space] = 0
```

B.4. On Packet Sent

Whenever a packet is sent, and it contains non-ACK frames, the packet increases bytes_in_flight.

```
OnPacketSentCC(sent_bytes):
    bytes_in_flight += sent_bytes
```

B.5. On Packet Acknowledgment

Invoked from loss detection's OnAckReceived and is supplied with the newly acked_packets from sent_packets.

In congestion avoidance, implementers that use an integer representation for congestion_window should be careful with division, and can use the alternative approach suggested in Section 2.1 of [RFC3465].

```
InCongestionRecovery(sent_time):
    return sent_time <= congestion_recovery_start_time

OnPacketsAacked(acked_packets):
    for acked_packet in acked_packets:
        OnPacketAacked(acked_packet)

OnPacketAacked(acked_packet):
    if (!acked_packet.in_flight):
        return;
    // Remove from bytes_in_flight.
    bytes_in_flight -= acked_packet.sent_bytes
    // Do not increase congestion_window if application
    // limited or flow control limited.
    if (IsAppOrFlowControlLimited())
        return
    // Do not increase congestion window in recovery period.
    if (InCongestionRecovery(acked_packet.time_sent)):
        return
    if (congestion_window < ssthresh):
        // Slow start.
        congestion_window += acked_packet.sent_bytes
    else:
        // Congestion avoidance.
        congestion_window +=
            max_datagram_size * acked_packet.sent_bytes
        / congestion_window
```

B.6. On New Congestion Event

Invoked from ProcessECN and OnPacketsLost when a new congestion event is detected. If not already in recovery, this starts a recovery period and reduces the slow start threshold and congestion window immediately.

```
OnCongestionEvent(sent_time):
    // No reaction if already in a recovery period.
    if (InCongestionRecovery(sent_time)):
        return

    // Enter recovery period.
    congestion_recovery_start_time = now()
    ssthresh = congestion_window * kLossReductionFactor
    congestion_window = max(ssthresh, kMinimumWindow)
    // A packet can be sent to speed up loss recovery.
    MaybeSendOnePacket()
```


B.7. Process ECN Information

Invoked when an ACK frame with an ECN section is received from the peer.

```
ProcessECN(ack, pn_space):
    // If the ECN-CE counter reported by the peer has increased,
    // this could be a new congestion event.
    if (ack.ce_counter > ecn_ce_counters[pn_space]):
        ecn_ce_counters[pn_space] = ack.ce_counter
        sent_time = sent_packets[ack.largest_acked].time_sent
        OnCongestionEvent(sent_time)
```

B.8. On Packets Lost

Invoked when DetectAndRemoveLostPackets deems packets lost.

```
OnPacketsLost(lost_packets):
    sent_time_of_last_loss = 0
    // Remove lost packets from bytes_in_flight.
    for lost_packet in lost_packets:
        if lost_packet.in_flight:
            bytes_in_flight -= lost_packet.sent_bytes
            sent_time_of_last_loss =
                max(sent_time_of_last_loss, lost_packet.time_sent)
    // Congestion event if in-flight packets were lost
    if (sent_time_of_last_loss != 0):
        OnCongestionEvent(sent_time_of_last_loss)

    // Reset the congestion window if the loss of these
    // packets indicates persistent congestion.
    // Only consider packets sent after getting an RTT sample.
    if (first_rtt_sample == 0):
        return
    pc_lost = []
    for lost in lost_packets:
        if lost.time_sent > first_rtt_sample:
            pc_lost.insert(lost)
    if (InPersistentCongestion(pc_lost)):
        congestion_window = kMinimumWindow
        congestion_recovery_start_time = 0
```

B.9. Removing Discarded Packets From Bytes In Flight

When Initial or Handshake keys are discarded, packets sent in that space no longer count toward bytes in flight.

Pseudocode for RemoveFromBytesInFlight follows:

```
RemoveFromBytesInFlight(discarded_packets):  
    // Remove any unacknowledged packets from flight.  
    foreach packet in discarded_packets:  
        if packet.in_flight  
            bytes_in_flight -= size
```

Appendix C. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-recovery-32

- * Clarifications to definition of persistent congestion (#4413, #4414, #4421, #4429, #4437)

C.2. Since draft-ietf-quic-recovery-31

- * Limit the number of Initial packets sent in response to unauthenticated packets (#4183, #4188)

C.3. Since draft-ietf-quic-recovery-30

Editorial changes only.

C.4. Since draft-ietf-quic-recovery-29

- * Allow caching of packets that can't be decrypted, by allowing the reported acknowledgment delay to exceed max_ack_delay prior to confirming the handshake (#3821, #3980, #4035, #3874)
- * Persistent congestion cannot include packets sent before the first RTT sample for the path (#3875, #3889)
- * Recommend reset of min_rtt in persistent congestion (#3927, #3975)
- * Persistent congestion is independent of packet number space (#3939, #3961)
- * Only limit bursts to the initial window without information about the path (#3892, #3936)
- * Add normative requirements for increasing and reducing the congestion window (#3944, #3978, #3997, #3998)

C.5. Since draft-ietf-quic-recovery-28

- * Refactored pseudocode to correct PTO calculation (#3564, #3674, #3681)

C.6. Since draft-ietf-quic-recovery-27

- * Added recommendations for speeding up handshake under some loss conditions (#3078, #3080)
- * PTO count is reset when handshake progress is made (#3272, #3415)
- * PTO count is not reset by a client when the server might be awaiting address validation (#3546, #3551)
- * Recommend repairing losses immediately after entering the recovery period (#3335, #3443)
- * Clarified what loss conditions can be ignored during the handshake (#3456, #3450)
- * Allow, but don't recommend, using RTT from previous connection to seed RTT (#3464, #3496)
- * Recommend use of adaptive loss detection thresholds (#3571, #3572)

C.7. Since draft-ietf-quic-recovery-26

No changes.

C.8. Since draft-ietf-quic-recovery-25

No significant changes.

C.9. Since draft-ietf-quic-recovery-24

- * Require congestion control of some sort (#3247, #3244, #3248)
- * Set a minimum reordering threshold (#3256, #3240)
- * PTO is specific to a packet number space (#3067, #3074, #3066)

C.10. Since draft-ietf-quic-recovery-23

- * Define under-utilizing the congestion window (#2630, #2686, #2675)
- * PTO MUST send data if possible (#3056, #3057)

- * Connection Close is not ack-eliciting (#3097, #3098)
- * MUST limit bursts to the initial congestion window (#3160)
- * Define the current max_datagram_size for congestion control (#3041, #3167)

C.11. Since draft-ietf-quic-recovery-22

- * PTO should always send an ack-eliciting packet (#2895)
- * Unify the Handshake Timer with the PTO timer (#2648, #2658, #2886)
- * Move ACK generation text to transport draft (#1860, #2916)

C.12. Since draft-ietf-quic-recovery-21

- * No changes

C.13. Since draft-ietf-quic-recovery-20

- * Path validation can be used as initial RTT value (#2644, #2687)
- * max_ack_delay transport parameter defaults to 0 (#2638, #2646)
- * ACK delay only measures intentional delays induced by the implementation (#2596, #2786)

C.14. Since draft-ietf-quic-recovery-19

- * Change kPersistentThreshold from an exponent to a multiplier (#2557)
- * Send a PING if the PTO timer fires and there's nothing to send (#2624)
- * Set loss delay to at least kGranularity (#2617)
- * Merge application limited and sending after idle sections. Always limit burst size instead of requiring resetting CWND to initial CWND after idle (#2605)
- * Rewrite RTT estimation, allow RTT samples where a newly acked packet is ack-eliciting but the largest_acked is not (#2592)
- * Don't arm the handshake timer if there is no handshake data (#2590)

- * Clarify that the time threshold loss alarm takes precedence over the crypto handshake timer (#2590, #2620)
- * Change initial RTT to 500ms to align with RFC6298 (#2184)

C.15. Since draft-ietf-quic-recovery-18

- * Change IW byte limit to 14720 from 14600 (#2494)
- * Update PTO calculation to match RFC6298 (#2480, #2489, #2490)
- * Improve loss detection's description of multiple packet number spaces and pseudocode (#2485, #2451, #2417)
- * Declare persistent congestion even if non-probe packets are sent and don't make persistent congestion more aggressive than RTO verified was (#2365, #2244)
- * Move pseudocode to the appendices (#2408)
- * What to send on multiple PTOs (#2380)

C.16. Since draft-ietf-quic-recovery-17

- * After Probe Timeout discard in-flight packets or send another (#2212, #1965)
- * Endpoints discard initial keys as soon as handshake keys are available (#1951, #2045)
- * 0-RTT state is discarded when 0-RTT is rejected (#2300)
- * Loss detection timer is cancelled when ack-eliciting frames are in flight (#2117, #2093)
- * Packets are declared lost if they are in flight (#2104)
- * After becoming idle, either pace packets or reset the congestion controller (#2138, 2187)
- * Process ECN counts before marking packets lost (#2142)
- * Mark packets lost before resetting crypto_count and pto_count (#2208, #2209)
- * Congestion and loss recovery state are discarded when keys are discarded (#2327)

C.17. Since draft-ietf-quic-recovery-16

- * Unify TLP and RTO into a single PTO; eliminate min RTO, min TLP and min crypto timeouts; eliminate timeout validation (#2114, #2166, #2168, #1017)
- * Redefine how congestion avoidance in terms of when the period starts (#1928, #1930)
- * Document what needs to be tracked for packets that are in flight (#765, #1724, #1939)
- * Integrate both time and packet thresholds into loss detection (#1969, #1212, #934, #1974)
- * Reduce congestion window after idle, unless pacing is used (#2007, #2023)
- * Disable RTT calculation for packets that don't elicit acknowledgment (#2060, #2078)
- * Limit ack_delay by max_ack_delay (#2060, #2099)
- * Initial keys are discarded once Handshake keys are available (#1951, #2045)
- * Reorder ECN and loss detection in pseudocode (#2142)
- * Only cancel loss detection timer if ack-eliciting packets are in flight (#2093, #2117)

C.18. Since draft-ietf-quic-recovery-14

- * Used max_ack_delay from transport params (#1796, #1782)
- * Merge ACK and ACK_ECN (#1783)

C.19. Since draft-ietf-quic-recovery-13

- * Corrected the lack of ssthresh reduction in CongestionEvent pseudocode (#1598)
- * Considerations for ECN spoofing (#1426, #1626)
- * Clarifications for PADDING and congestion control (#837, #838, #1517, #1531, #1540)
- * Reduce early retransmission timer to RTT/8 (#945, #1581)

- * Packets are declared lost after an RTO is verified (#935, #1582)

C.20. Since draft-ietf-quic-recovery-12

- * Changes to manage separate packet number spaces and encryption levels (#1190, #1242, #1413, #1450)
- * Added ECN feedback mechanisms and handling; new ACK_ECN frame (#804, #805, #1372)

C.21. Since draft-ietf-quic-recovery-11

No significant changes.

C.22. Since draft-ietf-quic-recovery-10

- * Improved text on ack generation (#1139, #1159)
- * Make references to TCP recovery mechanisms informational (#1195)
- * Define time_of_last_sent_handshake_packet (#1171)
- * Added signal from TLS the data it includes needs to be sent in a Retry packet (#1061, #1199)
- * Minimum RTT (min_rtt) is initialized with an infinite value (#1169)

C.23. Since draft-ietf-quic-recovery-09

No significant changes.

C.24. Since draft-ietf-quic-recovery-08

- * Clarified pacing and RTO (#967, #977)

C.25. Since draft-ietf-quic-recovery-07

- * Include ACK delay in RTO(and TLP) computations (#981)
- * ACK delay in SRTT computation (#961)
- * Default RTT and Slow Start (#590)
- * Many editorial fixes.

C.26. Since draft-ietf-quic-recovery-06

No significant changes.

C.27. Since draft-ietf-quic-recovery-05

- * Add more congestion control text (#776)

C.28. Since draft-ietf-quic-recovery-04

No significant changes.

C.29. Since draft-ietf-quic-recovery-03

No significant changes.

C.30. Since draft-ietf-quic-recovery-02

- * Integrate F-RTO (#544, #409)

- * Add congestion control (#545, #395)

- * Require connection abort if a skipped packet was acknowledged (#415)

- * Simplify RTO calculations (#142, #417)

C.31. Since draft-ietf-quic-recovery-01

- * Overview added to loss detection

- * Changes initial default RTT to 100ms

- * Added time-based loss detection and fixes early retransmit

- * Clarified loss recovery for handshake packets

- * Fixed references and made TCP references informative

C.32. Since draft-ietf-quic-recovery-00

- * Improved description of constants and ACK behavior

C.33. Since draft-iyengar-quic-loss-recovery-01

- * Adopted as base for draft-ietf-quic-recovery

- * Updated authors/editors list

- * Added table of contents

Appendix D. Contributors

The IETF QUIC Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document:

- * Alessandro Ghedini
- * Benjamin Saunders
- * Gorrry Fairhurst
- * (Kazu Yamamoto)
- * (Kazuho Oku)
- * Lars Eggert
- * Magnus Westerlund
- * Marten Seemann
- * Martin Duke
- * Martin Thomson
- * Mirja Kühlewind
- * Nick Banks
- * Praveen Balasubramanian

Acknowledgments

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Ian Swett (editor)
Google

Email: ianswett@google.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd
March 13, 2017

Using Transport Layer Security (TLS) to Secure QUIC
draft-ietf-quic-tls-02

Abstract

This document describes how Transport Layer Security (TLS) can be used to secure QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at <https://github.com/quicwg> ; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/tls> .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Notational Conventions	4
3. Protocol Overview	4
3.1. TLS Overview	5
3.2. TLS Handshake	6
4. TLS Usage	7
4.1. Handshake and Setup Sequence	8
4.2. Interface to TLS	9
4.2.1. Handshake Interface	9
4.2.2. Source Address Validation	11
4.2.3. Key Ready Events	11
4.2.4. Secret Export	12
4.2.5. TLS Interface Summary	12
4.3. TLS Version	13
4.4. ClientHello Size	13
4.5. Peer Authentication	14
4.6. TLS Errors	14
5. QUIC Packet Protection	14
5.1. Installing New Keys	15
5.2. QUIC Key Expansion	15
5.2.1. 0-RTT Secret	15
5.2.2. 1-RTT Secrets	16
5.2.3. Packet Protection Key and IV	17
5.3. QUIC AEAD Usage	18
5.4. Packet Numbers	19
5.5. Receiving Protected Packets	19
6. Key Phases	20
6.1. Packet Protection for the TLS Handshake	20
6.1.1. Initial Key Transitions	21
6.1.2. Retransmission and Acknowledgment of Unprotected Packets	22
6.2. Key Update	22
7. Client Address Validation	24
7.1. HelloRetryRequest Address Validation	24
7.2. NewSessionTicket Address Validation	25
7.3. Address Validation Token Integrity	26

8.	Pre-handshake QUIC Messages	26
8.1.	Unprotected Packets Prior to Handshake Completion	27
8.1.1.	STREAM Frames	27
8.1.2.	ACK Frames	27
8.1.3.	WINDOW_UPDATE Frames	28
8.1.4.	Denial of Service with Unprotected Packets	28
8.2.	Use of 0-RTT Keys	29
8.3.	Receiving Out-of-Order Protected Frames	29
9.	QUIC-Specific Additions to the TLS Handshake	30
9.1.	Protocol and Version Negotiation	30
9.2.	QUIC Transport Parameters Extension	31
9.3.	Priming 0-RTT	31
10.	Security Considerations	32
10.1.	Packet Reflection Attack Mitigation	32
10.2.	Peer Denial of Service	32
11.	Error codes	33
12.	IANA Considerations	33
13.	References	33
13.1.	Normative References	33
13.2.	Informative References	34
Appendix A.	Contributors	35
Appendix B.	Acknowledgments	35
Appendix C.	Change Log	35
C.1.	Since draft-ietf-quic-tls-01:	35
C.2.	Since draft-ietf-quic-tls-00:	35
C.3.	Since draft-thomson-quic-tls-01:	36
Authors' Addresses	36

1. Introduction

QUIC [QUIC-TRANSPORT] provides a multiplexed transport. When used for HTTP [RFC7230] semantics [QUIC-HTTP] it provides several key advantages over HTTP/1.1 [RFC7230] or HTTP/2 [RFC7540] over TCP [RFC0793].

This document describes how QUIC can be secured using Transport Layer Security (TLS) version 1.3 [I-D.ietf-tls-tls13]. TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

This document describes how the standardized TLS 1.3 can act a security component of QUIC. The same design could work for TLS 1.2, though few of the benefits QUIC provides would be realized due to the handshake latency in versions of TLS prior to 1.3.

2. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

This document uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3.

TLS terminology is used when referring to parts of TLS. Though TLS assumes a continuous stream of octets, it divides that stream into `_records_`. Most relevant to QUIC are the records that contain TLS `_handshake messages_`, which are discrete messages that are used for key agreement, authentication and parameter negotiation. Ordinarily, TLS records can also contain `_application data_`, though in the QUIC usage there is no use of TLS application data.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS 1.3 connection [I-D.ietf-tls-tls13]; QUIC also relies on TLS 1.3 for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols are co-dependent: QUIC uses the TLS handshake; TLS uses the reliability and ordered delivery provided by QUIC streams.

This document defines how QUIC interacts with TLS. This includes a description of how TLS is used, how keying material is derived from TLS, and the application of that keying material to protect QUIC packets. Figure 1 shows the basic interactions between TLS and QUIC, with the QUIC packet protection being called out specially.

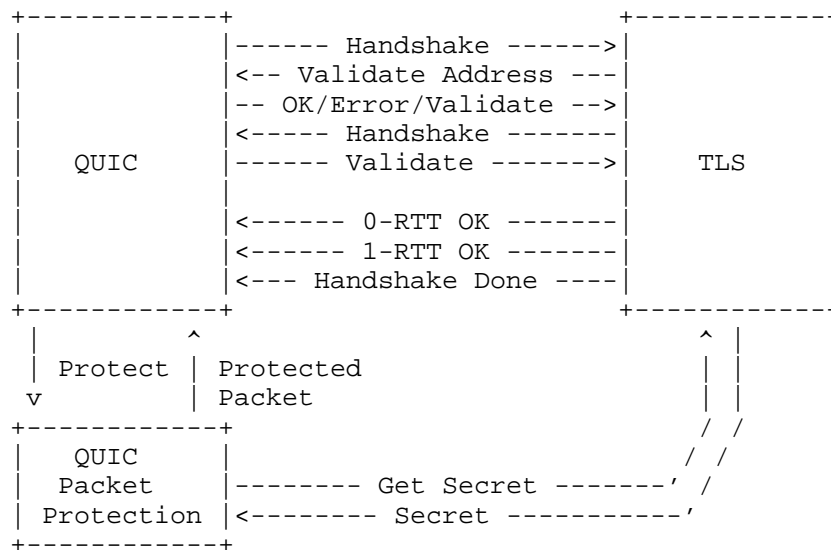


Figure 1: QUIC and TLS Interactions

The initial state of a QUIC connection has packets exchanged without any form of protection. In this state, QUIC is limited to using stream 1 and associated packets. Stream 1 is reserved for a TLS connection. This is a complete TLS connection as it would appear when layered over TCP; the only difference is that QUIC provides the reliability and ordering that would otherwise be provided by TCP.

At certain points during the TLS handshake, keying material is exported from the TLS connection for use by QUIC. This keying material is used to derive packet protection keys. Details on how and when keys are derived and used are included in Section 5.

This arrangement means that some TLS messages receive redundant protection from both the QUIC packet protection and the TLS record protection. These messages are limited in number; the TLS connection is rarely needed once the handshake completes.

3.1. TLS Overview

TLS provides two endpoints a way to establish a means of communication over an untrusted medium (that is, the Internet) that ensures that messages they exchange cannot be observed, modified, or forged.

TLS features can be separated into two basic functions: an authenticated key exchange and record protection. QUIC primarily

uses the authenticated key exchange provided by TLS but provides its own packet protection.

The TLS authenticated key exchange occurs between two entities: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman (DH) key exchanges. PSK is the basis for 0-RTT; the latter provides perfect forward secrecy (PFS) when the DH keys are destroyed.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 certificate-based authentication [RFC5280] for both server and client.

The TLS key exchange is resistant to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

3.2. TLS Handshake

TLS 1.3 provides two basic handshake modes of interest to QUIC:

- o A full, 1-RTT handshake in which the client is able to send application data after one round trip and the server immediately after receiving the first handshake message from the client.
- o A 0-RTT handshake in which the client uses information it has previously learned about the server to send immediately. This data can be replayed by an attacker so it MUST NOT carry a self-contained trigger for any non-idempotent action.

A simplified TLS 1.3 handshake with 0-RTT application data is shown in Figure 2, see [I-D.ietf-tls-tls13] for more options and details.

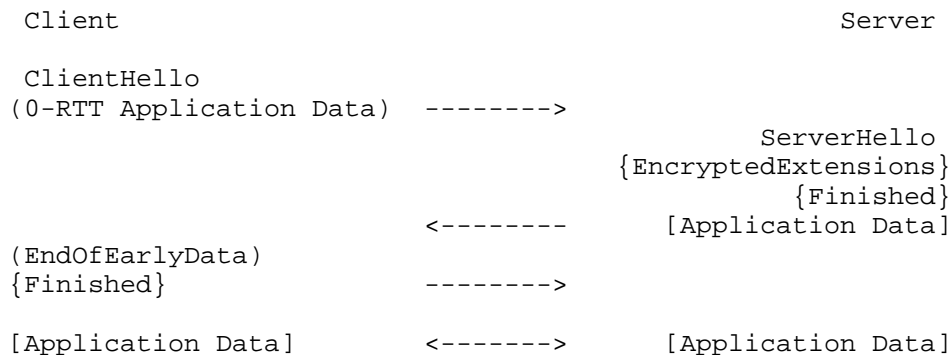


Figure 2: TLS Handshake with 0-RTT

This 0-RTT handshake is only possible if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected application data until it has received all of the handshake messages sent by the server.

Two additional variations on this basic handshake exchange are relevant to this document:

- o The server can respond to a ClientHello with a HelloRetryRequest, which adds an additional round trip prior to the basic exchange. This is needed if the server wishes to request a different key exchange key from the client. HelloRetryRequest is also used to verify that the client is correctly able to receive packets on the address it claims to have (see [QUIC-TRANSPORT]).
- o A pre-shared key mode can be used for subsequent handshakes to avoid public key operations. This is the basis for 0-RTT data, even if the remainder of the connection is protected by a new Diffie-Hellman exchange.

4. TLS Usage

QUIC reserves stream 1 for a TLS connection. Stream 1 contains a complete TLS connection, which includes the TLS record layer. Other than the definition of a QUIC-specific extension (see Section-TBD), TLS is unmodified for this use. This means that TLS will apply confidentiality and integrity protection to its records. In particular, TLS record protection is what provides confidentiality protection for the TLS handshake messages sent by the server.

QUIC permits a client to send frames on streams starting from the first packet. The initial packet from a client contains a stream frame for stream 1 that contains the first TLS handshake messages

from the client. This allows the TLS handshake to start with the first packet that a client sends.

QUIC packets are protected using a scheme that is specific to QUIC, see Section 5. Keys are exported from the TLS connection when they become available using a TLS exporter (see Section 7.3.3 of [I-D.ietf-tls-tls13] and Section 5.2). After keys are exported from TLS, QUIC manages its own key schedule.

4.1. Handshake and Setup Sequence

The integration of QUIC with a TLS handshake is shown in more detail in Figure 3. QUIC "STREAM" frames on stream 1 carry the TLS handshake. QUIC performs loss recovery [QUIC-RECOVERY] for this stream and ensures that TLS handshake messages are delivered in the correct order.

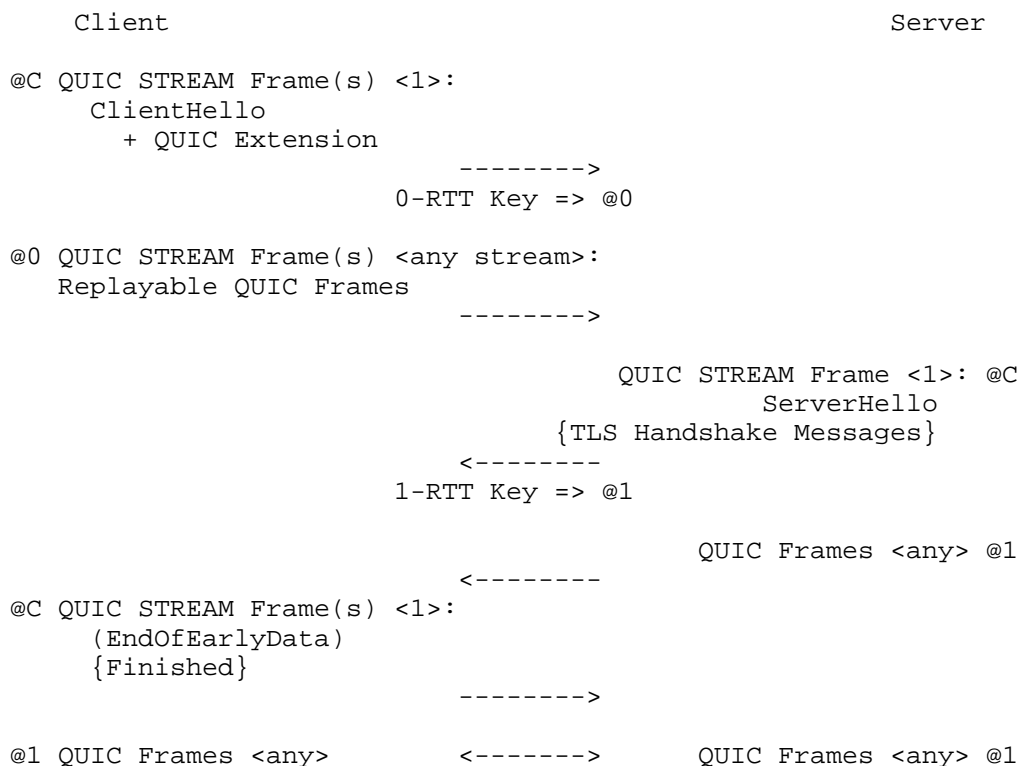


Figure 3: QUIC over TLS Handshake

In Figure 3, symbols mean:

- o "<" and ">" enclose stream numbers.
- o "@" indicates the key phase that is currently used for protecting QUIC packets.
- o "(" and ")" enclose messages that are protected with TLS 0-RTT handshake or application keys.
- o "{" and "}" enclose messages that are protected by the TLS Handshake keys.

If 0-RTT is not attempted, then the client does not send packets protected by the 0-RTT key (@0). In that case, the only key transition on the client is from unprotected packets (@C) to 1-RTT protection (@1), which happens after it sends its final set of TLS handshake messages.

The server sends TLS handshake messages without protection (@C). The server transitions from no protection (@C) to full 1-RTT protection (@1) after it sends the last of its handshake messages.

Some TLS handshake messages are protected by the TLS handshake record protection. These keys are not exported from the TLS connection for use in QUIC. QUIC packets from the server are sent in the clear until the final transition to 1-RTT keys.

The client transitions from cleartext (@C) to 0-RTT keys (@0) when sending 0-RTT data, and subsequently to 1-RTT keys (@1) after its second flight of TLS handshake messages. This creates the potential for unprotected packets to be received by a server in close proximity to packets that are protected with 1-RTT keys.

More information on key transitions is included in Section 6.1.

4.2. Interface to TLS

As shown in Figure 1, the interface from QUIC to TLS consists of four primary functions: Handshake, Source Address Validation, Key Ready Events, and Secret Export.

Additional functions might be needed to configure TLS.

4.2.1. Handshake Interface

In order to drive the handshake, TLS depends on being able to send and receive handshake messages on stream 1. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides handshake packets.

Before starting the handshake QUIC provides TLS with the transport parameters (see Section 9.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake octets from TLS. The client acquires handshake octets before sending its first packet.

A QUIC server starts the process by providing TLS with stream 1 octets.

Each time that an endpoint receives data on stream 1, it delivers the octets to TLS if it is able. Each time that TLS is provided with new data, new handshake octets are requested from TLS. TLS might not provide any octets if the handshake messages it has received are incomplete or it has no data to send.

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake octets that TLS needs to send. TLS also provides QUIC with the transport parameters that the peer advertised during the handshake.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested - either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives on stream 1. In the same way that is done during the handshake, new data is requested from TLS after providing received data.

Important: Until the handshake is reported as complete, the connection and key exchange are not properly authenticated at the server. Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, the server cannot consider the client to be authenticated until it receives and validates the client's Finished message.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending a copy of the STREAM frame that carries the Finished message in multiple packets. This enables immediate server processing for those packets.

4.2.2. Source Address Validation

During the processing of the TLS ClientHello, TLS requests that the transport make a decision about whether to request source address validation from the client.

An initial TLS ClientHello that resumes a session includes an address validation token in the session ticket; this includes all attempts at 0-RTT. If the client does not attempt session resumption, no token will be present. While processing the initial ClientHello, TLS provides QUIC with any token that is present. In response, QUIC provides one of three responses:

- o proceed with the connection,
- o ask for client address validation, or
- o abort the connection.

If QUIC requests source address validation, it also provides a new address validation token. TLS includes that along with any information it requires in the cookie extension of a TLS HelloRetryRequest message. In the other cases, the connection either proceeds or terminates with a handshake error.

The client echoes the cookie extension in a second ClientHello. A ClientHello that contains a valid cookie extension will be always be in response to a HelloRetryRequest. If address validation was requested by QUIC, then this will include an address validation token. TLS makes a second address validation request of QUIC, including the value extracted from the cookie extension. In response to this request, QUIC cannot ask for client address validation, it can only abort or permit the connection attempt to proceed.

QUIC can provide a new address validation token for use in session resumption at any time after the handshake is complete. Each time a new token is provided TLS generates a NewSessionTicket message, with the token included in the ticket.

See Section 7 for more details on client address validation.

4.2.3. Key Ready Events

TLS provides QUIC with signals when 0-RTT and 1-RTT keys are ready for use. These events are not asynchronous, they always occur immediately after TLS is provided with new handshake octets, or after TLS produces handshake octets.

When TLS completed its handshake, 1-RTT keys can be provided to QUIC. On both client and server, this occurs after sending the TLS Finished message.

This ordering means that there could be frames that carry TLS handshake messages ready to send at the same time that application data is available. An implementation **MUST** ensure that TLS handshake messages are always sent in cleartext packets. Separate packets are required for data that needs protection from 1-RTT keys.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake octets, the TLS stack might signal that 0-RTT keys are ready. On the server, after receiving handshake octets that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

1-RTT keys are used for packets in both directions. 0-RTT keys are only used to protect packets sent by the client.

4.2.4. Secret Export

Details how secrets are exported from TLS are included in Section 5.2.

4.2.5. TLS Interface Summary

Figure 4 summarizes the exchange between QUIC and TLS for both client and server.

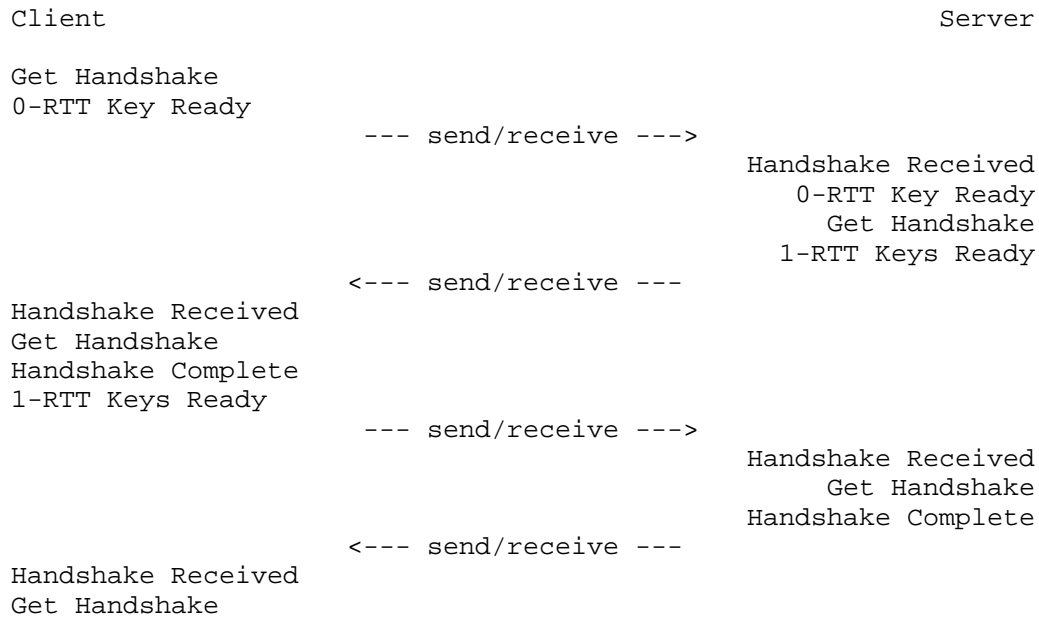


Figure 4: Interaction Summary between QUIC and TLS

4.3. TLS Version

This document describes how TLS 1.3 [I-D.ietf-tls-tls13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a newer version of TLS than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint **MUST** terminate the connection if a version of TLS older than 1.3 is negotiated.

4.4. ClientHello Size

QUIC requires that the initial handshake packet from a client fit within a single packet of at least 1280 octets. With framing and packet overheads this value could be reduced.

A TLS ClientHello can fit within this limit with ample space remaining. However, there are several variables that could cause this limit to be exceeded. Implementations are reminded that large

session tickets or HelloRetryRequest cookies, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, the size of the session tickets and HelloRetryRequest cookie extension can have an effect on a client's ability to connect. Choosing a small value increases the probability that these values can be successfully used by a client.

A TLS implementation does not need to enforce this size constraint. QUIC padding can be used to reach this size, meaning that a TLS server is unlikely to receive a large ClientHello message.

4.5. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client **MUST** authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

A server **MAY** request that the client authenticate during the handshake. A server **MAY** refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server **MUST NOT** use post-handshake client authentication (see Section 4.6.2 of [I-D.ietf-tls-tls13]).

4.6. TLS Errors

Errors in the TLS connection **SHOULD** be signaled using TLS alerts on stream 1. A failure in the handshake **MUST** be treated as a QUIC connection error of type `TLS_HANDSHAKE_FAILED`. Once the handshake is complete, an error in the TLS connection that causes a TLS alert to be sent or received **MUST** be treated as a QUIC connection error of type `TLS_FATAL_ALERT_GENERATED` or `TLS_FATAL_ALERT_RECEIVED` respectively.

5. QUIC Packet Protection

QUIC packet protection provides authenticated encryption of packets. This provides confidentiality and integrity protection for the

content of packets (see Section 5.3). Packet protection uses keys that are exported from the TLS connection (see Section 5.2).

Different keys are used for QUIC packet protection and TLS record protection. Having separate QUIC and TLS record protection means that TLS records can be protected by two different keys. This redundancy is limited to only a few TLS records, and is maintained for the sake of simplicity.

5.1. Installing New Keys

As TLS reports the availability of keying material, the packet protection keys and initialization vectors (IVs) are updated (see Section 5.2). The selection of AEAD function is also updated to match the AEAD negotiated by TLS.

For packets other than any unprotected handshake packets (see Section 6.1), once a change of keys has been made, packets with higher packet numbers **MUST** use the new keying material. The `KEY_PHASE` bit on these packets is inverted each time new keys are installed to signal the use of the new keys to the recipient (see Section 6 for details).

An endpoint retransmits stream data in a new packet. New packets have new packet numbers and use the latest packet protection keys. This simplifies key management when there are key updates (see Section 6.2).

5.2. QUIC Key Expansion

QUIC uses a system of packet protection secrets, keys and IVs that are modelled on the system used in TLS [I-D.ietf-tls-tls13]. The secrets that QUIC uses as the basis of its key schedule are obtained using TLS exporters (see Section 7.3.3 of [I-D.ietf-tls-tls13]).

QUIC uses HKDF with the same hash function negotiated by TLS for key derivation. For example, if TLS is using the `TLS_AES_128_GCM_SHA256`, the SHA-256 hash function is used.

5.2.1. 0-RTT Secret

0-RTT keys are those keys that are used in resumed connections prior to the completion of the TLS handshake. Data sent using 0-RTT keys might be replayed and so has some restrictions on its use, see Section 8.2. 0-RTT keys are used after sending or receiving a `ClientHello`.

The secret is exported from TLS using the exporter label "EXPORTER-QUIC 0-RTT Secret" and an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS. This uses the TLS `early_exporter_secret`. The QUIC 0-RTT secret is only used for protection of packets sent by the client.

```
client_0rtt_secret
  = TLS-Exporter("EXPORTER-QUIC 0-RTT Secret"
    "", Hash.length)
```

5.2.2. 1-RTT Secrets

1-RTT keys are used by both client and server after the TLS handshake completes. There are two secrets used at any time: one is used to derive packet protection keys for packets sent by the client, the other for packet protection keys on packets sent by the server.

The initial client packet protection secret is exported from TLS using the exporter label "EXPORTER-QUIC client 1-RTT Secret"; the initial server packet protection secret uses the exporter label "EXPORTER-QUIC server 1-RTT Secret". Both exporters use an empty context. The size of the secret MUST be the size of the hash output for the PRF hash function negotiated by TLS.

```
client_pp_secret_0
  = TLS-Exporter("EXPORTER-QUIC client 1-RTT Secret"
    "", Hash.length)
server_pp_secret_0
  = TLS-Exporter("EXPORTER-QUIC server 1-RTT Secret"
    "", Hash.length)
```

These secrets are used to derive the initial client and server packet protection keys.

After a key update (see Section 6.2), these secrets are updated using the HKDF-Expand-Label function defined in Section 7.1 of [I-D.ietf-tls-tls13]. HKDF-Expand-Label uses the PRF hash function negotiated by TLS. The replacement secret is derived using the existing Secret, a Label of "QUIC client 1-RTT Secret" for the client and "QUIC server 1-RTT Secret" for the server, an empty HashValue, and the same output Length as the hash function selected by TLS for its PRF.

```

client_pp_secret_<N+1>
    = HKDF-Expand-Label(client_pp_secret_<N>,
                        "QUIC client 1-RTT Secret",
                        "", Hash.length)
server_pp_secret_<N+1>
    = HKDF-Expand-Label(server_pp_secret_<N>,
                        "QUIC server 1-RTT Secret",
                        "", Hash.length)

```

This allows for a succession of new secrets to be created as needed.

HKDF-Expand-Label uses HKDF-Expand [RFC5869] with a specially formatted info parameter. The info parameter that includes the output length (in this case, the size of the PRF hash output) encoded on two octets in network byte order, the length of the prefixed Label as a single octet, the value of the Label prefixed with "TLS 1.3, ", and a zero octet to indicate an empty HashValue. For example, the client packet protection secret uses an info parameter of:

```

info = (HashLen / 256) || (HashLen % 256) || 0x21 ||
       "TLS 1.3, QUIC client 1-RTT secret" || 0x00

```

5.2.3. Packet Protection Key and IV

The complete key expansion uses an identical process for key expansion as defined in Section 7.3 of [I-D.ietf-tls-tls13], using different values for the input secret. QUIC uses the AEAD function negotiated by TLS.

The packet protection key and IV used to protect the 0-RTT packets sent by a client use the QUIC 0-RTT secret. This uses the HKDF-Expand-Label with the PRF hash function negotiated by TLS.

The length of the output is determined by the requirements of the AEAD function selected by TLS. The key length is the AEAD key size. As defined in Section 5.3 of [I-D.ietf-tls-tls13], the IV length is the larger of 8 or N_MIN (see Section 4 of [RFC5116]).

```

client_0rtt_key = HKDF-Expand-Label(client_0rtt_secret,
                                    "key", "", key_length)
client_0rtt_iv = HKDF-Expand-Label(client_0rtt_secret,
                                    "iv", "", iv_length)

```

Similarly, the packet protection key and IV used to protect 1-RTT packets sent by both client and server use the current packet protection secret.

```
client_pp_key_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                     "key", "", key_length)
client_pp_iv_<N> = HKDF-Expand-Label(client_pp_secret_<N>,
                                     "iv", "", iv_length)
server_pp_key_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                     "key", "", key_length)
server_pp_iv_<N> = HKDF-Expand-Label(server_pp_secret_<N>,
                                     "iv", "", iv_length)
```

The client protects (or encrypts) packets with the client packet protection key and IV; the server protects packets with the server packet protection key.

The QUIC record protection initially starts without keying material. When the TLS state machine reports that the ClientHello has been sent, the 0-RTT keys can be generated and installed for writing. When the TLS state machine reports completion of the handshake, the 1-RTT keys can be generated and installed for writing.

5.3. QUIC AEAD Usage

The Authentication Encryption with Associated Data (AEAD) [RFC5116] function used for QUIC packet protection is AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256, the AEAD_AES_128_GCM function is used.

Regular QUIC packets are protected by an AEAD [RFC5116]. Version negotiation and public reset packets are not protected.

Once TLS has provided a key, the contents of regular QUIC packets immediately after any TLS messages have been sent are protected by the AEAD selected by TLS.

The key, *K*, for the AEAD is either the client packet protection key (*client_pp_key_n*) or the server packet protection key (*server_pp_key_n*), derived as defined in Section 5.2.

The nonce, *N*, for the AEAD is formed by combining either the packet protection IV (either *client_pp_iv_n* or *server_pp_iv_n*) with packet numbers. The 64 bits of the reconstructed QUIC packet number in network byte order is left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, *A*, for the AEAD is the contents of the QUIC header, starting from the flags octet in the common header.

The input plaintext, *P*, for the AEAD is the contents of the QUIC frame following the packet number, as described in [QUIC-TRANSPORT].

The output ciphertext, *C*, of the AEAD is transmitted in place of *P*.

Prior to TLS providing keys, no record protection is performed and the plaintext, *P*, is transmitted unmodified.

5.4. Packet Numbers

QUIC has a single, contiguous packet number space. In comparison, TLS restarts its sequence number each time that record protection keys are changed. The sequence number restart in TLS ensures that a compromise of the current traffic keys does not allow an attacker to truncate the data that is sent after a key update by sending additional packets under the old key (causing new packets to be discarded).

QUIC does not assume a reliable transport and is required to handle attacks where packets are dropped in other ways. QUIC is therefore not affected by this form of truncation.

The QUIC packet number is not reset and it is not permitted to go higher than its maximum value of $2^{64}-1$. This establishes a hard limit on the number of packets that can be sent.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV (see for example [AEBounds]). This might be lower than the packet number limit. An endpoint **MUST** initiate a key update (Section 6.2) prior to exceeding any limit set for the AEAD that is in use.

TLS maintains a separate sequence number that is used for record protection on the connection that is hosted on stream 1. This sequence number is not visible to QUIC.

5.5. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it **MUST** discard all packets with higher packet numbers if they cannot be successfully unprotected with either the same key, or - if there is a key update - the next packet protection key (see Section 6.2). Similarly, a packet that appears to trigger a key update, but cannot be unprotected successfully **MUST** be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated

packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

6. Key Phases

As TLS reports the availability of 0-RTT and 1-RTT keys, new keying material can be exported from TLS and used for QUIC packet protection. At each transition during the handshake a new secret is exported from TLS and packet protection keys are derived from that secret.

Every time that a new set of keys is used for protecting outbound packets, the KEY_PHASE bit in the public flags is toggled. The exception is the transition from 0-RTT keys to 1-RTT keys, where the presence of the version field and its associated bit is used (see Section 6.1.1).

Once the connection is fully enabled, the KEY_PHASE bit allows a recipient to detect a change in keying material without necessarily needing to receive the first packet that triggered the change. An endpoint that notices a changed KEY_PHASE bit can update keys and decrypt the packet that contains the changed bit, see Section 6.2.

The KEY_PHASE bit is the third bit of the public flags (0x04).

Transitions between keys during the handshake are complicated by the need to ensure that TLS handshake messages are sent with the correct packet protection.

6.1. Packet Protection for the TLS Handshake

The initial exchange of packets are sent without protection. These packets are marked with a KEY_PHASE of 0.

TLS handshake messages MUST NOT be protected using QUIC packet protection. A KEY_PHASE of 0 is used for all of these packets, even during retransmission. The messages affected are all TLS handshake message up to the TLS Finished that is sent by each endpoint.

Any TLS handshake messages that are sent after completing the TLS handshake do not need special packet protection rules. Packets containing these messages use the packet protection keys that are current at the time of sending (or retransmission).

Like the client, a server MUST send retransmissions of its unprotected handshake messages or acknowledgments for unprotected handshake messages sent by the client in unprotected packets (KEY_PHASE=0).

6.1.1. Initial Key Transitions

Once the TLS handshake is complete, keying material is exported from TLS and QUIC packet protection commences.

Packets protected with 1-RTT keys have a KEY_PHASE bit set to 1. These packets also have a VERSION bit set to 0.

If the client sends 0-RTT data, it marks packets protected with 0-RTT keys with a KEY_PHASE of 1 and a VERSION bit of 1. Setting the version bit means that all packets also include the version field. The client retains the VERSION bit, but reverts the KEY_PHASE bit for the packet that contains the TLS EndOfEarlyData and Finished messages.

The client clears the VERSION bit and sets the KEY_PHASE bit to 1 when it transitions to using 1-RTT keys.

Marking 0-RTT data with the both KEY_PHASE and VERSION bits ensures that the server is able to identify these packets as 0-RTT data in case packets containing TLS handshake message are lost or delayed. Including the version also ensures that the packet format is known to the server in this case.

Using both KEY_PHASE and VERSION also ensures that the server is able to distinguish between cleartext handshake packets (KEY_PHASE=0, VERSION=1), 0-RTT protected packets (KEY_PHASE=1, VERSION=1), and 1-RTT protected packets (KEY_PHASE=1, VERSION=0). Packets with all of these markings can arrive concurrently, and being able to identify each cleanly ensures that the correct packet protection keys can be selected and applied.

A server might choose to retain 0-RTT packets that arrive before a TLS ClientHello. The server can then use those packets once the ClientHello arrives. However, the potential for denial of service from buffering 0-RTT packets is significant. These packets cannot be authenticated and so might be employed by an attacker to exhaust server resources. Limiting the number of packets that are saved might be necessary.

The server transitions to using 1-RTT keys after sending its first flight of TLS handshake messages. From this point, the server protects all packets with 1-RTT keys. Future packets are therefore protected with 1-RTT keys and marked with a KEY_PHASE of 1.

6.1.2. Retransmission and Acknowledgment of Unprotected Packets

TLS handshake messages from both client and server are critical to the key exchange. The contents of these messages determines the keys used to protect later messages. If these handshake messages are included in packets that are protected with these keys, they will be indecipherable to the recipient.

Even though newer keys could be available when retransmitting, retransmissions of these handshake messages **MUST** be sent in unprotected packets (with a `KEY_PHASE` of 0). An endpoint **MUST** also generate ACK frames for these messages that are sent in unprotected packets.

A `HelloRetryRequest` handshake message might be used to reject an initial `ClientHello`. A `HelloRetryRequest` handshake message and any second `ClientHello` that is sent in response **MUST** also be sent without packet protection. This is natural, because no new keying material will be available when these messages need to be sent. Upon receipt of a `HelloRetryRequest`, a client **SHOULD** cease any transmission of 0-RTT data; 0-RTT data will only be discarded by any server that sends a `HelloRetryRequest`.

The `KEY_PHASE` and `VERSION` bits ensure that protected packets are clearly distinguished from unprotected packets. Loss or reordering might cause unprotected packets to arrive once 1-RTT keys are in use, unprotected packets are easily distinguished from 1-RTT packets.

Once 1-RTT keys are available to an endpoint, it no longer needs the TLS handshake messages that are carried in unprotected packets. However, a server might need to retransmit its TLS handshake messages in response to receiving an unprotected packet that contains ACK frames. A server **MUST** process ACK frames in unprotected packets until the TLS handshake is reported as complete, or it receives an ACK frame in a protected packet that acknowledges all of its handshake messages.

To limit the number of key phases that could be active, an endpoint **MUST NOT** initiate a key update while there are any unacknowledged handshake messages, see Section 6.2.

6.2. Key Update

Once the TLS handshake is complete, the `KEY_PHASE` bit allows for refreshes of keying material by either peer. Endpoints start using updated keys immediately without additional signaling; the change in the `KEY_PHASE` bit indicates that a new key is in use.

An endpoint MUST NOT initiate more than one key update at a time. A new key cannot be used until the endpoint has received and successfully decrypted a packet with a matching KEY_PHASE. Note that when 0-RTT is attempted the value of the KEY_PHASE bit will be different on packets sent by either peer.

A receiving endpoint detects an update when the KEY_PHASE bit doesn't match what it is expecting. It creates a new secret (see Section 5.2) and the corresponding read key and IV. If the packet can be decrypted and authenticated using these values, then the keys it uses for packet protection are also updated. The next packet sent by the endpoint will then use the new keys.

An endpoint doesn't need to send packets immediately when it detects that its peer has updated keys. The next packet that it sends will simply use the new keys. If an endpoint detects a second update before it has sent any packets with updated keys it indicates that its peer has updated keys twice without awaiting a reciprocal update. An endpoint MUST treat consecutive key updates as a fatal error and abort the connection.

An endpoint SHOULD retain old keys for a short period to allow it to decrypt packets with smaller packet numbers than the packet that triggered the key update. This allows an endpoint to consume packets that are reordered around the transition between keys. Packets with higher packet numbers always use the updated keys and MUST NOT be decrypted with old keys.

Keys and their corresponding secrets SHOULD be discarded when an endpoint has received all packets with sequence numbers lower than the lowest sequence number used for the new key. An endpoint might discard keys if it determines that the length of the delay to affected packets is excessive.

This ensures that once the handshake is complete, packets with the same KEY_PHASE will have the same packet protection keys, unless there are multiple key updates in a short time frame succession and significant packet reordering.

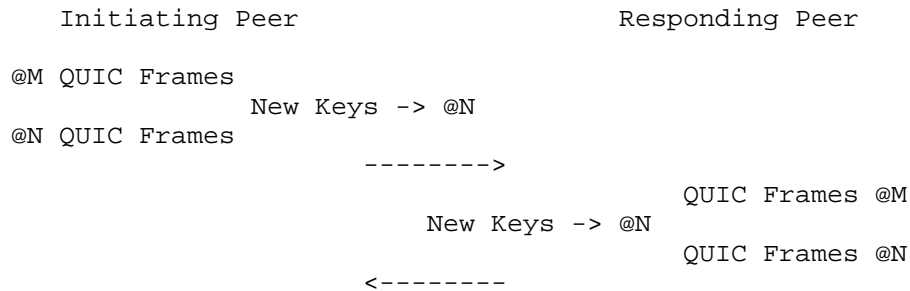


Figure 5: Key Update

As shown in Figure 3 and Figure 5, there is never a situation where there are more than two different sets of keying material that might be received by a peer. Once both sending and receiving keys have been updated,

A server cannot initiate a key update until it has received the client's Finished message. Otherwise, packets protected by the updated keys could be confused for retransmissions of handshake messages. A client cannot initiate a key update until all of its handshake messages have been acknowledged by the server.

A packet that triggers a key update could arrive after successfully processing a packet with a higher packet number. This is only possible if there is a key compromise and an attack, or if the peer is incorrectly reverting to use of old keys. Because the latter cannot be differentiated from an attack, an endpoint MUST immediately terminate the connection if it detects this condition.

7. Client Address Validation

Two tools are provided by TLS to enable validation of client source addresses at a server: the cookie in the HelloRetryRequest message, and the ticket in the NewSessionTicket message.

7.1. HelloRetryRequest Address Validation

The cookie extension in the TLS HelloRetryRequest message allows a server to perform source address validation during the handshake.

When QUIC requests address validation during the processing of the first ClientHello, the token it provides is included in the cookie extension of a HelloRetryRequest. As long as the cookie cannot be successfully guessed by a client, the server can be assured that the client received the HelloRetryRequest if it includes the value in a second ClientHello.

An initial ClientHello never includes a cookie extension. Thus, if a server constructs a cookie that contains all the information necessary to reconstruct state, it can discard local state after sending a HelloRetryRequest. Presence of a valid cookie in a ClientHello indicates that the ClientHello is a second attempt from the client.

An address validation token can be extracted from a second ClientHello and passed to the transport for further validation. If that validation fails, the server **MUST** fail the TLS handshake and send an `illegal_parameter` alert.

Combining address validation with the other uses of HelloRetryRequest ensures that there are fewer ways in which an additional round-trip can be added to the handshake. In particular, this makes it possible to combine a request for address validation with a request for a different client key share.

If TLS needs to send a HelloRetryRequest for other reasons, it needs to ensure that it can correctly identify the reason that the HelloRetryRequest was generated. During the processing of a second ClientHello, TLS does not need to consult the transport protocol regarding address validation if address validation was not requested originally. In such cases, the cookie extension could either be absent or it could indicate that an address validation token is not present.

7.2. NewSessionTicket Address Validation

The ticket in the TLS NewSessionTicket message allows a server to provide a client with a similar sort of token. When a client resumes a TLS connection - whether or not 0-RTT is attempted - it includes the ticket in the handshake message. As with the HelloRetryRequest cookie, the server includes the address validation token in the ticket. TLS provides the token it extracts from the session ticket to the transport when it asks whether source address validation is needed.

If both a HelloRetryRequest cookie and a session ticket are present in the ClientHello, only the token from the cookie is passed to the transport. The presence of a cookie indicates that this is a second ClientHello - the token from the session ticket will have been provided to the transport when it appeared in the first ClientHello.

A server can send a NewSessionTicket message at any time. This allows it to update the state - and the address validation token - that is included in the ticket. This might be done to refresh the ticket or token, or it might be generated in response to changes in

the state of the connection. QUIC can request that a NewSessionTicket be sent by providing a new address validation token.

A server that intends to support 0-RTT SHOULD provide an address validation token immediately after completing the TLS handshake.

7.3. Address Validation Token Integrity

TLS MUST provide integrity protection for address validation token unless the transport guarantees integrity protection by other means. For a NewSessionTicket that includes confidential information - such as the resumption secret - including the token under authenticated encryption ensures that the token gains both confidentiality and integrity protection without duplicating the overheads of that protection.

8. Pre-handshake QUIC Messages

Implementations MUST NOT exchange data on any stream other than stream 1 without packet protection. QUIC requires the use of several types of frame for managing loss detection and recovery during this phase. In addition, it might be useful to use the data acquired during the exchange of unauthenticated messages for congestion control.

This section generally only applies to TLS handshake messages from both peers and acknowledgments of the packets carrying those messages. In many cases, the need for servers to provide acknowledgments is minimal, since the messages that clients send are small and implicitly acknowledged by the server's responses.

The actions that a peer takes as a result of receiving an unauthenticated packet needs to be limited. In particular, state established by these packets cannot be retained once record protection commences.

There are several approaches possible for dealing with unauthenticated packets prior to handshake completion:

- o discard and ignore them
- o use them, but reset any state that is established once the handshake completes
- o use them and authenticate them afterwards; failing the handshake if they can't be authenticated
- o save them and use them when they can be properly authenticated

- o treat them as a fatal error

Different strategies are appropriate for different types of data. This document proposes that all strategies are possible depending on the type of message.

- o Transport parameters are made usable and authenticated as part of the TLS handshake (see Section 9.2).
- o Most unprotected messages are treated as fatal errors when received except for the small number necessary to permit the handshake to complete (see Section 8.1).
- o Protected packets can either be discarded or saved and later used (see Section 8.3).

8.1. Unprotected Packets Prior to Handshake Completion

This section describes the handling of messages that are sent and received prior to the completion of the TLS handshake.

Sending and receiving unprotected messages is hazardous. Unless expressly permitted, receipt of an unprotected message of any kind MUST be treated as a fatal error.

8.1.1. STREAM Frames

"STREAM" frames for stream 1 are permitted. These carry the TLS handshake messages. Once 1-RTT keys are available, unprotected "STREAM" frames on stream 1 can be ignored.

Receiving unprotected "STREAM" frames for other streams MUST be treated as a fatal error.

8.1.2. ACK Frames

"ACK" frames are permitted prior to the handshake being complete. Information learned from "ACK" frames cannot be entirely relied upon, since an attacker is able to inject these packets. Timing and packet retransmission information from "ACK" frames is critical to the functioning of the protocol, but these frames might be spoofed or altered.

Endpoints MUST NOT use an unprotected "ACK" frame to acknowledge data that was protected by 0-RTT or 1-RTT keys. An endpoint MUST ignore an unprotected "ACK" frame if it claims to acknowledge data that was sent in a protected packet. Such an acknowledgement can only serve

as a denial of service, since an endpoint that can read protected data is always able to send protected data.

ISSUE: What about 0-RTT data? Should we allow acknowledgment of 0-RTT with unprotected frames? If we don't, then 0-RTT data will be unacknowledged until the handshake completes. This isn't a problem if the handshake completes without loss, but it could mean that 0-RTT stalls when a handshake packet disappears for any reason.

An endpoint SHOULD use data from unprotected or 0-RTT-protected "ACK" frames only during the initial handshake and while they have insufficient information from 1-RTT-protected "ACK" frames. Once sufficient information has been obtained from protected messages, information obtained from less reliable sources can be discarded.

8.1.3. WINDOW_UPDATE Frames

"WINDOW_UPDATE" frames MUST NOT be sent unprotected.

Though data is exchanged on stream 1, the initial flow control window is sufficiently large to allow the TLS handshake to complete. This limits the maximum size of the TLS handshake and would prevent a server or client from using an abnormally large certificate chain.

Stream 1 is exempt from the connection-level flow control window.

8.1.4. Denial of Service with Unprotected Packets

Accepting unprotected - specifically unauthenticated - packets presents a denial of service risk to endpoints. An attacker that is able to inject unprotected packets can cause a recipient to drop even protected packets with a matching sequence number. The spurious packet shadows the genuine packet, causing the genuine packet to be ignored as redundant.

Once the TLS handshake is complete, both peers MUST ignore unprotected packets. From that point onward, unprotected messages can be safely dropped.

Since only TLS handshake packets and acknowledgments are sent in the clear, an attacker is able to force implementations to rely on retransmission for packets that are lost or shadowed. Thus, an attacker that intends to deny service to an endpoint has to drop or shadow protected packets in order to ensure that their victim continues to accept unprotected packets. The ability to shadow packets means that an attacker does not need to be on path.

ISSUE: This would not be an issue if QUIC had a randomized starting sequence number. If we choose to randomize, we fix this problem and reduce the denial of service exposure to on-path attackers. The only possible problem is in authenticating the initial value, so that peers can be sure that they haven't missed an initial message.

In addition to causing valid packets to be dropped, an attacker can generate packets with an intent of causing the recipient to expend processing resources. See Section 10.2 for a discussion of these risks.

To avoid receiving TLS packets that contain no useful data, a TLS implementation **MUST** reject empty TLS handshake records and any record that is not permitted by the TLS state machine. Any TLS application data or alerts that is received prior to the end of the handshake **MUST** be treated as a fatal error.

8.2. Use of 0-RTT Keys

If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client **MUST** only use 0-RTT keys to protect data that is idempotent. A client **MAY** wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client **SHOULD** stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server **MUST NOT** use 0-RTT keys to protect packets.

8.3. Receiving Out-of-Order Protected Frames

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client.

Packets protected with 1-RTT keys **MAY** be stored and later decrypted and used once the handshake is complete. A server **MUST NOT** use 1-RTT protected packets before verifying either the client Finished message or - in the case that the server has chosen to use a pre-shared key -

the pre-shared key binder (see Section 4.2.8 of [I-D.ietf-tls-tls13]). Verifying these values provides the server with an assurance that the ClientHello has not been modified.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server MAY retain these packets for later decryption in anticipation of receiving a ClientHello.

Receiving and verifying the TLS Finished message is critical in ensuring the integrity of the TLS handshake. A server MUST NOT use protected packets from the client prior to verifying the client Finished message if its response depends on client authentication.

9. QUIC-Specific Additions to the TLS Handshake

QUIC uses the TLS handshake for more than just negotiation of cryptographic parameters. The TLS handshake validates protocol version selection, provides preliminary values for QUIC transport parameters, and allows a server to perform return routeability checks on clients.

9.1. Protocol and Version Negotiation

The QUIC version negotiation mechanism is used to negotiate the version of QUIC that is used prior to the completion of the handshake. However, this packet is not authenticated, enabling an active attacker to force a version downgrade.

To ensure that a QUIC version downgrade is not forced by an attacker, version information is copied into the TLS handshake, which provides integrity protection for the QUIC negotiation. This does not prevent version downgrade during the handshake, though it means that such a downgrade causes a handshake failure.

TLS uses Application Layer Protocol Negotiation (ALPN) [RFC7301] to select an application protocol. The application-layer protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected.

If the server cannot select a compatible combination of application protocol and QUIC version, it MUST abort the connection. A client MUST abort a connection if the server picks an incompatible combination of QUIC version and ALPN identifier.

9.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different format for this struct.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {  
    quic_transport_parameters(26), (65535)  
} ExtensionType;
```

The "extension_data" field of the quic_transport_parameters extension contains a value that is defined by the version of QUIC that is in use. The quic_transport_parameters extension carries a TransportParameters when the version of QUIC defined in [QUIC-TRANSPORT] is used.

9.3. Priming 0-RTT

QUIC uses TLS without modification. Therefore, it is possible to use a pre-shared key that was obtained in a TLS connection over TCP to enable 0-RTT in QUIC. Similarly, QUIC can provide a pre-shared key that can be used to enable 0-RTT in TCP.

All the restrictions on the use of 0-RTT apply, with the exception of the ALPN label, which MUST only change to a label that is explicitly designated as being compatible. The client indicates which ALPN label it has chosen by placing that ALPN label first in the ALPN extension.

The certificate that the server uses MUST be considered valid for both connections, which will use different protocol stacks and could use different port numbers. For instance, HTTP/1.1 and HTTP/2 operate over TLS and TCP, whereas QUIC operates over UDP.

Source address validation is not completely portable between different protocol stacks. Even if the source IP address remains constant, the port number is likely to be different. Packet reflection attacks are still possible in this situation, though the set of hosts that can initiate these attacks is greatly reduced. A server might choose to avoid source address validation for such a connection, or allow an increase to the amount of data that it sends toward the client without source validation.

10. Security Considerations

There are likely to be some real clangers here eventually, but the current set of issues is well captured in the relevant sections of the main text.

Never assume that because it isn't in the security considerations section it doesn't affect security. Most of this document does.

10.1. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

Certificate caching [RFC7924] can reduce the size of the server's handshake messages significantly.

QUIC requires that the packet containing a ClientHello be padded to the size of the maximum transmission unit (MTU). A server is less likely to generate a packet reflection attack if the data it sends is a small multiple of this size. A server SHOULD use a HelloRetryRequest if the size of the handshake messages it sends is likely to significantly exceed the size of the packet containing the ClientHello.

10.2. Peer Denial of Service

QUIC, TLS and HTTP/2 all contain a messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection. If processing is disproportionately large in comparison to the observable effects on bandwidth or state, then this could allow a malicious peer to exhaust processing capacity without consequence.

QUIC prohibits the sending of empty "STREAM" frames unless they are marked with the FIN bit. This prevents "STREAM" frames from being sent that only waste effort.

TLS records SHOULD always contain at least one octet of a handshake messages or alert. Records containing only padding are permitted during the handshake, but an excessive number might be used to generate unnecessary work. Once the TLS handshake is complete, endpoints SHOULD NOT send TLS application data records unless it is to hide the length of QUIC records. QUIC packet protection does not include any allowance for padding; padded TLS application data records can be used to mask the length of QUIC frames.

While there are legitimate uses for some redundant packets, implementations SHOULD track redundant packets and treat excessive volumes of any non-productive packets as indicative of an attack.

11. Error codes

The portion of the QUIC error code space allocated for the crypto handshake is 0xC0000000-0xFFFFFFFF. The following error codes are defined when TLS is used for the crypto handshake:

TLS_HANDSHAKE_FAILED (0xC000001C): The TLS handshake failed.

TLS_FATAL_ALERT_GENERATED (0xC000001D): A TLS fatal alert was sent, causing the TLS connection to end prematurely.

TLS_FATAL_ALERT_RECEIVED (0xC000001E): A TLS fatal alert was received, causing the TLS connection to end prematurely.

12. IANA Considerations

This document has no IANA actions. Yet.

13. References

13.1. Normative References

- [I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.
- [QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport".
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<http://www.rfc-editor.org/info/rfc5116>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<http://www.rfc-editor.org/info/rfc5869>>.

- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<http://www.rfc-editor.org/info/rfc7230>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.

13.2. Informative References

- [AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", March 2016, <<http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>>.
- [QUIC-HTTP] Bishop, M., Ed., "Hypertext Transfer Protocol (HTTP) over QUIC".
- [QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<http://www.rfc-editor.org/info/rfc793>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<http://www.rfc-editor.org/info/rfc2818>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<http://www.rfc-editor.org/info/rfc5280>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<http://www.rfc-editor.org/info/rfc7540>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<http://www.rfc-editor.org/info/rfc7924>>.

Appendix A. Contributors

Ryan Hamilton was originally an author of this specification.

Appendix B. Acknowledgments

This document has benefited from input from Dragana Damjanovic, Christian Huitema, Jana Iyengar, Adam Langley, Roberto Peon, Eric Rescorla, Ian Swett, and many others.

Appendix C. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-tls-01:

- o Use TLS alerts to signal TLS errors (#272, #374)
- o Require ClientHello to fit in a single packet (#338)
- o The second client handshake flight is now sent in the clear (#262, #337)
- o The QUIC header is included as AEAD Associated Data (#226, #243, #302)
- o Add interface necessary for client address validation (#275)
- o Define peer authentication (#140)
- o Require at least TLS 1.3 (#138)
- o Define transport parameters as a TLS extension (#122)
- o Define handling for protected packets before the handshake completes (#39)
- o Decouple QUIC version and ALPN (#12)

C.2. Since draft-ietf-quic-tls-00:

- o Changed bit used to signal key phase.
- o Updated key phase markings during the handshake.

- o Added TLS interface requirements section.
- o Moved to use of TLS exporters for key derivation.
- o Moved TLS error code definitions into this document.

C.3. Since draft-thomson-quic-tls-01:

- o Adopted as base for draft-ietf-quic-tls.
- o Updated authors/editors list.
- o Added status note.

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

Sean Turner (editor)
sn3rd

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 19 July 2021

M. Thomson, Ed.
Mozilla
S. Turner, Ed.
sn3rd
15 January 2021

Using TLS to Secure QUIC
draft-ietf-quic-tls-34

Abstract

This document describes how Transport Layer Security (TLS) is used to secure QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic.

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-tls>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 July 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
2. Notational Conventions	4
2.1. TLS Overview	5
3. Protocol Overview	7
4. Carrying TLS Messages	8
4.1. Interface to TLS	9
4.1.1. Handshake Complete	10
4.1.2. Handshake Confirmed	10
4.1.3. Sending and Receiving Handshake Messages	10
4.1.4. Encryption Level Changes	12
4.1.5. TLS Interface Summary	14
4.2. TLS Version	15
4.3. ClientHello Size	15
4.4. Peer Authentication	16
4.5. Session Resumption	17
4.6. 0-RTT	18
4.6.1. Enabling 0-RTT	18
4.6.2. Accepting and Rejecting 0-RTT	19
4.6.3. Validating 0-RTT Configuration	19
4.7. HelloRetryRequest	20
4.8. TLS Errors	20
4.9. Discarding Unused Keys	20
4.9.1. Discarding Initial Keys	21
4.9.2. Discarding Handshake Keys	21
4.9.3. Discarding 0-RTT Keys	22
5. Packet Protection	22
5.1. Packet Protection Keys	23
5.2. Initial Secrets	23
5.3. AEAD Usage	25
5.4. Header Protection	26
5.4.1. Header Protection Application	26
5.4.2. Header Protection Sample	28
5.4.3. AES-Based Header Protection	30
5.4.4. ChaCha20-Based Header Protection	30
5.5. Receiving Protected Packets	31
5.6. Use of 0-RTT Keys	31
5.7. Receiving Out-of-Order Protected Packets	32

5.8. Retry Packet Integrity	33
6. Key Update	34
6.1. Initiating a Key Update	36
6.2. Responding to a Key Update	37
6.3. Timing of Receive Key Generation	37
6.4. Sending with Updated Keys	38
6.5. Receiving with Different Keys	38
6.6. Limits on AEAD Usage	39
6.7. Key Update Error Code	41
7. Security of Initial Messages	41
8. QUIC-Specific Adjustments to the TLS Handshake	41
8.1. Protocol Negotiation	42
8.2. QUIC Transport Parameters Extension	42
8.3. Removing the EndOfEarlyData Message	43
8.4. Prohibit TLS Middlebox Compatibility Mode	43
9. Security Considerations	44
9.1. Session Linkability	44
9.2. Replay Attacks with 0-RTT	44
9.3. Packet Reflection Attack Mitigation	45
9.4. Header Protection Analysis	45
9.5. Header Protection Timing Side-Channels	46
9.6. Key Diversity	47
9.7. Randomness	47
10. IANA Considerations	47
11. References	48
11.1. Normative References	48
11.2. Informative References	49
Appendix A. Sample Packet Protection	51
A.1. Keys	51
A.2. Client Initial	52
A.3. Server Initial	54
A.4. Retry	55
A.5. ChaCha20-Poly1305 Short Header Packet	55
Appendix B. AEAD Algorithm Analysis	57
B.1. Analysis of AEAD_AES_128_GCM and AEAD_AES_256_GCM Usage Limits	58
B.1.1. Confidentiality Limit	58
B.1.2. Integrity Limit	58
B.2. Analysis of AEAD_AES_128_CCM Usage Limits	59
Appendix C. Change Log	60
C.1. Since draft-ietf-quic-tls-32	60
C.2. Since draft-ietf-quic-tls-31	60
C.3. Since draft-ietf-quic-tls-30	60
C.4. Since draft-ietf-quic-tls-29	60
C.5. Since draft-ietf-quic-tls-28	61
C.6. Since draft-ietf-quic-tls-27	61
C.7. Since draft-ietf-quic-tls-26	61
C.8. Since draft-ietf-quic-tls-25	61

C.9. Since draft-ietf-quic-tls-24	61
C.10. Since draft-ietf-quic-tls-23	61
C.11. Since draft-ietf-quic-tls-22	62
C.12. Since draft-ietf-quic-tls-21	62
C.13. Since draft-ietf-quic-tls-20	62
C.14. Since draft-ietf-quic-tls-18	62
C.15. Since draft-ietf-quic-tls-17	62
C.16. Since draft-ietf-quic-tls-14	62
C.17. Since draft-ietf-quic-tls-13	63
C.18. Since draft-ietf-quic-tls-12	63
C.19. Since draft-ietf-quic-tls-11	63
C.20. Since draft-ietf-quic-tls-10	63
C.21. Since draft-ietf-quic-tls-09	63
C.22. Since draft-ietf-quic-tls-08	63
C.23. Since draft-ietf-quic-tls-07	63
C.24. Since draft-ietf-quic-tls-05	64
C.25. Since draft-ietf-quic-tls-04	64
C.26. Since draft-ietf-quic-tls-03	64
C.27. Since draft-ietf-quic-tls-02	64
C.28. Since draft-ietf-quic-tls-01	64
C.29. Since draft-ietf-quic-tls-00	64
C.30. Since draft-thomson-quic-tls-01	65
Contributors	65
Authors' Addresses	66

1. Introduction

This document describes how QUIC [QUIC-TRANSPORT] is secured using TLS [TLS13].

TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

This document describes how TLS acts as a security component of QUIC.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the terminology established in [QUIC-TRANSPORT].

For brevity, the acronym TLS is used to refer to TLS 1.3, though a newer version could be used; see Section 4.2.

2.1. TLS Overview

TLS provides two endpoints with a way to establish a means of communication over an untrusted medium (for example, the Internet). TLS enables authentication of peers and provides confidentiality and integrity protection for messages that endpoints exchange.

Internally, TLS is a layered protocol, with the structure shown in Figure 1.

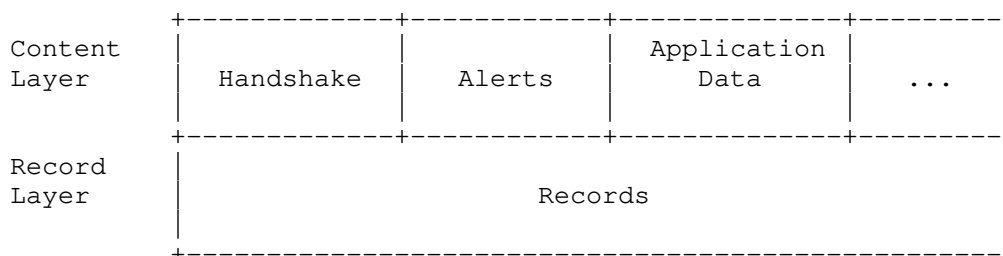


Figure 1: TLS Layers

Each Content layer message (e.g., Handshake, Alerts, and Application Data) is carried as a series of typed TLS records by the Record layer. Records are individually cryptographically protected and then transmitted over a reliable transport (typically TCP), which provides sequencing and guaranteed delivery.

The TLS authenticated key exchange occurs between two endpoints: client and server. The client initiates the exchange and the server responds. If the key exchange completes successfully, both client and server will agree on a secret. TLS supports both pre-shared key (PSK) and Diffie-Hellman over either finite fields or elliptic curves ((EC)DHE) key exchanges. PSK is the basis for Early Data (0-RTT); the latter provides forward secrecy (FS) when the (EC)DHE keys are destroyed. The two modes can also be combined, to provide forward secrecy while using the PSK for authentication.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 [RFC5280] certificate-based authentication for both server and client. When PSK key exchange is used (as in resumption), knowledge of the PSK serves to authenticate the peer.

The TLS key exchange is resistant to tampering by attackers and it produces shared secrets that cannot be controlled by either participating peer.

TLS provides two basic handshake modes of interest to QUIC:

- * A full 1-RTT handshake, in which the client is able to send Application Data after one round trip and the server immediately responds after receiving the first handshake message from the client.
- * A 0-RTT handshake, in which the client uses information it has previously learned about the server to send Application Data immediately. This Application Data can be replayed by an attacker so 0-RTT is not suitable for carrying instructions that might initiate any action that could cause unwanted effects if replayed.

A simplified TLS handshake with 0-RTT application data is shown in Figure 2.

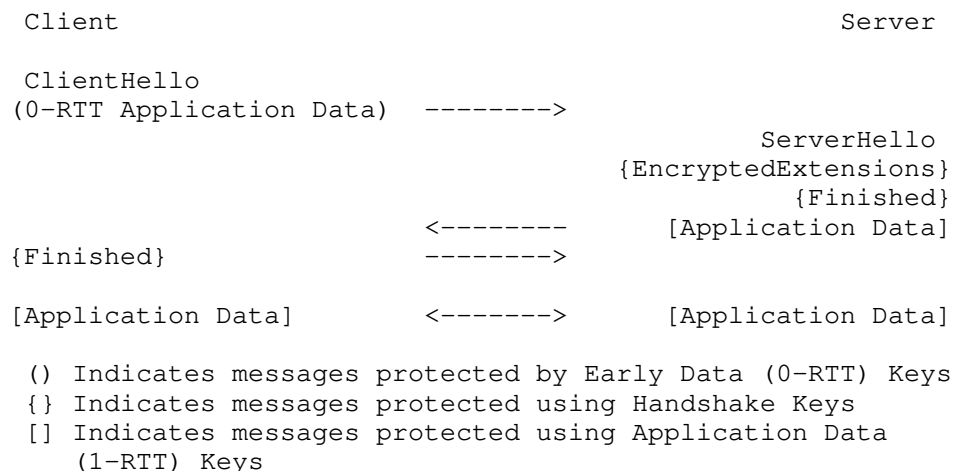


Figure 2: TLS Handshake with 0-RTT

Figure 2 omits the EndOfEarlyData message, which is not used in QUIC; see Section 8.3. Likewise, neither ChangeCipherSpec nor KeyUpdate messages are used by QUIC. ChangeCipherSpec is redundant in TLS 1.3; see Section 8.4. QUIC has its own key update mechanism; see Section 6.

Data is protected using a number of encryption levels:

- * Initial Keys

- * Early Data (0-RTT) Keys
- * Handshake Keys
- * Application Data (1-RTT) Keys

Application Data may appear only in the Early Data and Application Data levels. Handshake and Alert messages may appear in any level.

The 0-RTT handshake can be used if the client and server have previously communicated. In the 1-RTT handshake, the client is unable to send protected Application Data until it has received all of the Handshake messages sent by the server.

3. Protocol Overview

QUIC [QUIC-TRANSPORT] assumes responsibility for the confidentiality and integrity protection of packets. For this it uses keys derived from a TLS handshake [TLS13], but instead of carrying TLS records over QUIC (as with TCP), TLS Handshake and Alert messages are carried directly over the QUIC transport, which takes over the responsibilities of the TLS record layer, as shown in Figure 3.

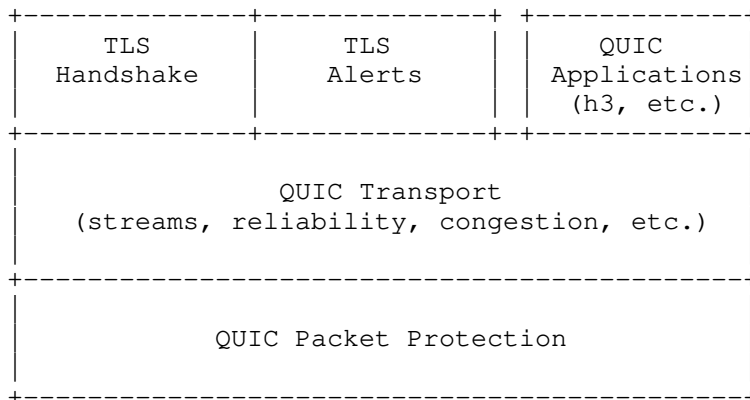


Figure 3: QUIC Layers

QUIC also relies on TLS for authentication and negotiation of parameters that are critical to security and performance.

Rather than a strict layering, these two protocols cooperate: QUIC uses the TLS handshake; TLS uses the reliability, ordered delivery, and record layer provided by QUIC.

At a high level, there are two main interactions between the TLS and QUIC components:

- * The TLS component sends and receives messages via the QUIC component, with QUIC providing a reliable stream abstraction to TLS.
- * The TLS component provides a series of updates to the QUIC component, including (a) new packet protection keys to install (b) state changes such as handshake completion, the server certificate, etc.

Figure 4 shows these interactions in more detail, with the QUIC packet protection being called out specially.

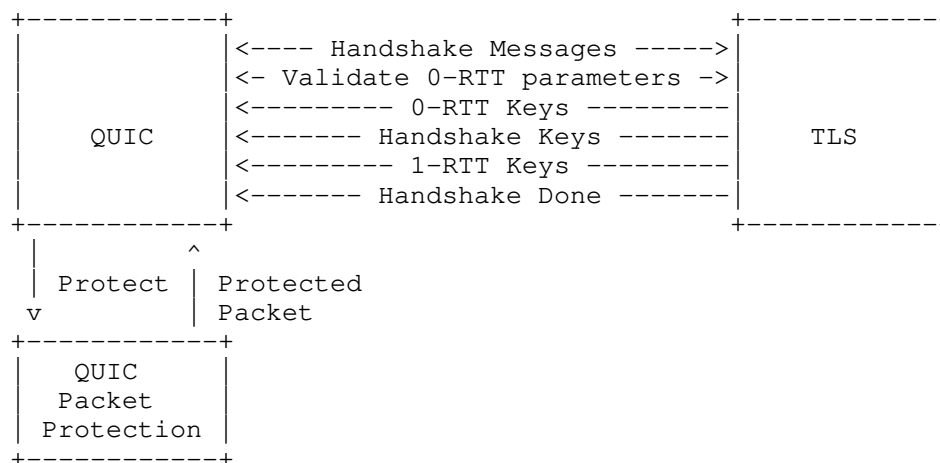


Figure 4: QUIC and TLS Interactions

Unlike TLS over TCP, QUIC applications that want to send data do not send it through TLS "application_data" records. Rather, they send it as QUIC STREAM frames or other frame types, which are then carried in QUIC packets.

4. Carrying TLS Messages

QUIC carries TLS handshake data in CRYPTO frames, each of which consists of a contiguous block of handshake data identified by an offset and length. Those frames are packaged into QUIC packets and encrypted under the current encryption level. As with TLS over TCP, once TLS handshake data has been delivered to QUIC, it is QUIC's responsibility to deliver it reliably. Each chunk of data that is produced by TLS is associated with the set of keys that TLS is

currently using. If QUIC needs to retransmit that data, it **MUST** use the same keys even if TLS has already updated to newer keys.

Each encryption level corresponds to a packet number space. The packet number space that is used determines the semantics of frames. Some frames are prohibited in different packet number spaces; see Section 12.5 of [QUIC-TRANSPORT].

Because packets could be reordered on the wire, QUIC uses the packet type to indicate which keys were used to protect a given packet, as shown in Table 1. When packets of different types need to be sent, endpoints **SHOULD** use coalesced packets to send them in the same UDP datagram.

Packet Type	Encryption Keys	PN Space
Initial	Initial secrets	Initial
0-RTT Protected	0-RTT	Application data
Handshake	Handshake	Handshake
Retry	Retry	N/A
Version Negotiation	N/A	N/A
Short Header	1-RTT	Application data

Table 1: Encryption Keys by Packet Type

Section 17 of [QUIC-TRANSPORT] shows how packets at the various encryption levels fit into the handshake process.

4.1. Interface to TLS

As shown in Figure 4, the interface from QUIC to TLS consists of four primary functions:

- * Sending and receiving handshake messages
- * Processing stored transport and application state from a resumed session and determining if it is valid to generate or accept early data
- * Rekeying (both transmit and receive)

* Handshake state updates

Additional functions might be needed to configure TLS. In particular, QUIC and TLS need to agree on which is responsible for validation of peer credentials, such as certificate validation ([RFC5280]).

4.1.1. Handshake Complete

In this document, the TLS handshake is considered complete when the TLS stack has reported that the handshake is complete. This happens when the TLS stack has both sent a Finished message and verified the peer's Finished message. Verifying the peer's Finished provides the endpoints with an assurance that previous handshake messages have not been modified. Note that the handshake does not complete at both endpoints simultaneously. Consequently, any requirement that is based on the completion of the handshake depends on the perspective of the endpoint in question.

4.1.2. Handshake Confirmed

In this document, the TLS handshake is considered confirmed at the server when the handshake completes. The server **MUST** send a HANDSHAKE_DONE frame as soon as the handshake is complete. At the client, the handshake is considered confirmed when a HANDSHAKE_DONE frame is received.

Additionally, a client **MAY** consider the handshake to be confirmed when it receives an acknowledgment for a 1-RTT packet. This can be implemented by recording the lowest packet number sent with 1-RTT keys, and comparing it to the Largest Acknowledged field in any received 1-RTT ACK frame: once the latter is greater than or equal to the former, the handshake is confirmed.

4.1.3. Sending and Receiving Handshake Messages

In order to drive the handshake, TLS depends on being able to send and receive handshake messages. There are two basic functions on this interface: one where QUIC requests handshake messages and one where QUIC provides bytes that comprise handshake messages.

Before starting the handshake QUIC provides TLS with the transport parameters (see Section 8.2) that it wishes to carry.

A QUIC client starts TLS by requesting TLS handshake bytes from TLS. The client acquires handshake bytes before sending its first packet. A QUIC server starts the process by providing TLS with the client's handshake bytes.

At any time, the TLS stack at an endpoint will have a current sending encryption level and receiving encryption level. TLS encryption levels determine the QUIC packet type and keys that are used for protecting data.

Each encryption level is associated with a different sequence of bytes, which is reliably transmitted to the peer in CRYPTO frames. When TLS provides handshake bytes to be sent, they are appended to the handshake bytes for the current encryption level. The encryption level then determines the type of packet that the resulting CRYPTO frame is carried in; see Table 1.

Four encryption levels are used, producing keys for Initial, 0-RTT, Handshake, and 1-RTT packets. CRYPTO frames are carried in just three of these levels, omitting the 0-RTT level. These four levels correspond to three packet number spaces: Initial and Handshake encrypted packets use their own separate spaces; 0-RTT and 1-RTT packets use the application data packet number space.

QUIC takes the unprotected content of TLS handshake records as the content of CRYPTO frames. TLS record protection is not used by QUIC. QUIC assembles CRYPTO frames into QUIC packets, which are protected using QUIC packet protection.

QUIC CRYPTO frames only carry TLS handshake messages. TLS alerts are turned into QUIC CONNECTION_CLOSE error codes; see Section 4.8. TLS application data and other content types cannot be carried by QUIC at any encryption level; it is an error if they are received from the TLS stack.

When an endpoint receives a QUIC packet containing a CRYPTO frame from the network, it proceeds as follows:

- * If the packet uses the current TLS receiving encryption level, sequence the data into the input flow as usual. As with STREAM frames, the offset is used to find the proper location in the data sequence. If the result of this process is that new data is available, then it is delivered to TLS in order.
- * If the packet is from a previously installed encryption level, it MUST NOT contain data that extends past the end of previously received data in that flow. Implementations MUST treat any violations of this requirement as a connection error of type `PROTOCOL_VIOLATION`.
- * If the packet is from a new encryption level, it is saved for later processing by TLS. Once TLS moves to receiving from this encryption level, saved data can be provided to TLS. When TLS

provides keys for a higher encryption level, if there is data from a previous encryption level that TLS has not consumed, this MUST be treated as a connection error of type `PROTOCOL_VIOLATION`.

Each time that TLS is provided with new data, new handshake bytes are requested from TLS. TLS might not provide any bytes if the handshake messages it has received are incomplete or it has no data to send.

The content of CRYPTO frames might either be processed incrementally by TLS or buffered until complete messages or flights are available. TLS is responsible for buffering handshake bytes that have arrived in order. QUIC is responsible for buffering handshake bytes that arrive out of order or for encryption levels that are not yet ready. QUIC does not provide any means of flow control for CRYPTO frames; see Section 7.5 of [QUIC-TRANSPORT].

Once the TLS handshake is complete, this is indicated to QUIC along with any final handshake bytes that TLS needs to send. At this stage, the transport parameters that the peer advertised during the handshake are authenticated; see Section 8.2.

Once the handshake is complete, TLS becomes passive. TLS can still receive data from its peer and respond in kind, but it will not need to send more data unless specifically requested - either by an application or QUIC. One reason to send data is that the server might wish to provide additional or updated session tickets to a client.

When the handshake is complete, QUIC only needs to provide TLS with any data that arrives in CRYPTO streams. In the same manner that is used during the handshake, new data is requested from TLS after providing received data.

4.1.4. Encryption Level Changes

As keys at a given encryption level become available to TLS, TLS indicates to QUIC that reading or writing keys at that encryption level are available.

The availability of new keys is always a result of providing inputs to TLS. TLS only provides new keys after being initialized (by a client) or when provided with new handshake data.

However, a TLS implementation could perform some of its processing asynchronously. In particular, the process of validating a certificate can take some time. While waiting for TLS processing to complete, an endpoint SHOULD buffer received packets if they might be processed using keys that aren't yet available. These packets can be processed once keys are provided by TLS. An endpoint SHOULD continue to respond to packets that can be processed during this time.

After processing inputs, TLS might produce handshake bytes, keys for new encryption levels, or both.

TLS provides QUIC with three items as a new encryption level becomes available:

- * A secret
- * An Authenticated Encryption with Associated Data (AEAD) function
- * A Key Derivation Function (KDF)

These values are based on the values that TLS negotiates and are used by QUIC to generate packet and header protection keys; see Section 5 and Section 5.4.

If 0-RTT is possible, it is ready after the client sends a TLS ClientHello message or the server receives that message. After providing a QUIC client with the first handshake bytes, the TLS stack might signal the change to 0-RTT keys. On the server, after receiving handshake bytes that contain a ClientHello message, a TLS server might signal that 0-RTT keys are available.

Although TLS only uses one encryption level at a time, QUIC may use more than one level. For instance, after sending its Finished message (using a CRYPTO frame at the Handshake encryption level) an endpoint can send STREAM data (in 1-RTT encryption). If the Finished message is lost, the endpoint uses the Handshake encryption level to retransmit the lost message. Reordering or loss of packets can mean that QUIC will need to handle packets at multiple encryption levels. During the handshake, this means potentially handling packets at higher and lower encryption levels than the current encryption level used by TLS.

In particular, server implementations need to be able to read packets at the Handshake encryption level at the same time as the 0-RTT encryption level. A client could interleave ACK frames that are protected with Handshake keys with 0-RTT data and the server needs to process those acknowledgments in order to detect lost Handshake packets.

QUIC also needs access to keys that might not ordinarily be available to a TLS implementation. For instance, a client might need to acknowledge Handshake packets before it is ready to send CRYPTO frames at that encryption level. TLS therefore needs to provide keys to QUIC before it might produce them for its own use.

4.1.5. TLS Interface Summary

Figure 5 summarizes the exchange between QUIC and TLS for both client and server. Solid arrows indicate packets that carry handshake data; dashed arrows show where application data can be sent. Each arrow is tagged with the encryption level used for that transmission.

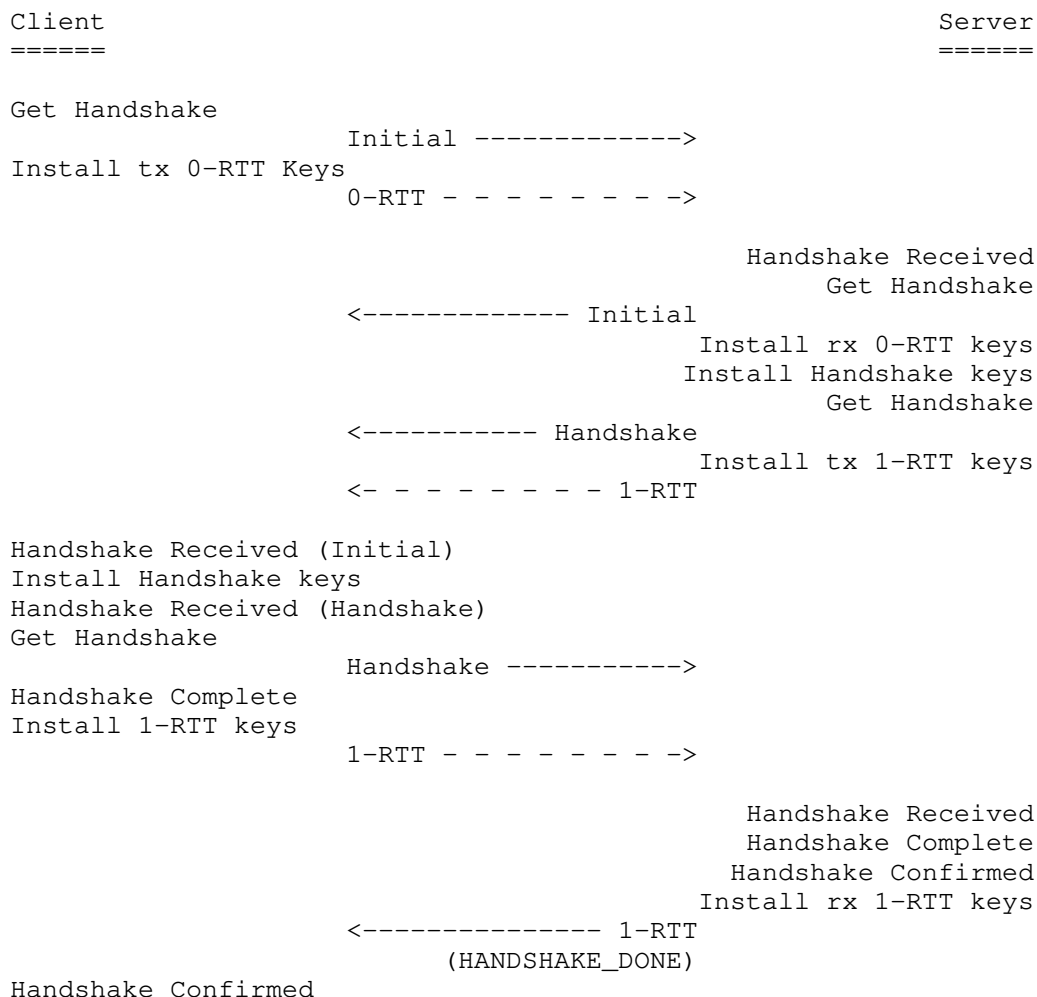


Figure 5: Interaction Summary between QUIC and TLS

Figure 5 shows the multiple packets that form a single "flight" of messages being processed individually, to show what incoming messages trigger different actions. This shows multiple "Get Handshake" invocations to retrieve handshake messages at different encryption levels. New handshake messages are requested after incoming packets have been processed.

Figure 5 shows one possible structure for a simple handshake exchange. The exact process varies based on the structure of endpoint implementations and the order in which packets arrive. Implementations could use a different number of operations or execute them in other orders.

4.2. TLS Version

This document describes how TLS 1.3 [TLS13] is used with QUIC.

In practice, the TLS handshake will negotiate a version of TLS to use. This could result in a newer version of TLS than 1.3 being negotiated if both endpoints support that version. This is acceptable provided that the features of TLS 1.3 that are used by QUIC are supported by the newer version.

Clients MUST NOT offer TLS versions older than 1.3. A badly configured TLS implementation could negotiate TLS 1.2 or another older version of TLS. An endpoint MUST terminate the connection if a version of TLS older than 1.3 is negotiated.

4.3. ClientHello Size

The first Initial packet from a client contains the start or all of its first cryptographic handshake message, which for TLS is the ClientHello. Servers might need to parse the entire ClientHello (e.g., to access extensions such as Server Name Identification (SNI) or Application Layer Protocol Negotiation (ALPN)) in order to decide whether to accept the new incoming QUIC connection. If the ClientHello spans multiple Initial packets, such servers would need to buffer the first received fragments, which could consume excessive resources if the client's address has not yet been validated. To avoid this, servers MAY use the Retry feature (see Section 8.1 of [QUIC-TRANSPORT]) to only buffer partial ClientHello messages from clients with a validated address.

QUIC packet and framing add at least 36 bytes of overhead to the ClientHello message. That overhead increases if the client chooses a source connection ID longer than zero bytes. Overheads also do not include the token or a destination connection ID longer than 8 bytes, both of which might be required if a server sends a Retry packet.

A typical TLS ClientHello can easily fit into a 1200-byte packet. However, in addition to the overheads added by QUIC, there are several variables that could cause this limit to be exceeded. Large session tickets, multiple or large key shares, and long lists of supported ciphers, signature algorithms, versions, QUIC transport parameters, and other negotiable parameters and extensions could cause this message to grow.

For servers, in addition to connection IDs and tokens, the size of TLS session tickets can have an effect on a client's ability to connect efficiently. Minimizing the size of these values increases the probability that clients can use them and still fit their entire ClientHello message in their first Initial packet.

The TLS implementation does not need to ensure that the ClientHello is large enough to meet the requirements for QUIC packets. QUIC PADDING frames are added to increase the size of the packet as necessary; see Section 14.1 of [QUIC-TRANSPORT].

4.4. Peer Authentication

The requirements for authentication depend on the application protocol that is in use. TLS provides server authentication and permits the server to request client authentication.

A client **MUST** authenticate the identity of the server. This typically involves verification that the identity of the server is included in a certificate and that the certificate is issued by a trusted entity (see for example [RFC2818]).

Note: Where servers provide certificates for authentication, the size of the certificate chain can consume a large number of bytes. Controlling the size of certificate chains is critical to performance in QUIC as servers are limited to sending 3 bytes for every byte received prior to validating the client address; see Section 8.1 of [QUIC-TRANSPORT]. The size of a certificate chain can be managed by limiting the number of names or extensions; using keys with small public key representations, like ECDSA; or by using certificate compression [COMPRESS].

A server MAY request that the client authenticate during the handshake. A server MAY refuse a connection if the client is unable to authenticate when requested. The requirements for client authentication vary based on application protocol and deployment.

A server MUST NOT use post-handshake client authentication (as defined in Section 4.6.2 of [TLS13]), because the multiplexing offered by QUIC prevents clients from correlating the certificate request with the application-level event that triggered it (see [HTTP2-TLS13]). More specifically, servers MUST NOT send post-handshake TLS CertificateRequest messages and clients MUST treat receipt of such messages as a connection error of type `PROTOCOL_VIOLATION`.

4.5. Session Resumption

QUIC can use the session resumption feature of TLS 1.3. It does this by carrying NewSessionTicket messages in CRYPTO frames after the handshake is complete. Session resumption can be used to provide 0-RTT, and can also be used when 0-RTT is disabled.

Endpoints that use session resumption might need to remember some information about the current connection when creating a resumed connection. TLS requires that some information be retained; see Section 4.6.1 of [TLS13]. QUIC itself does not depend on any state being retained when resuming a connection, unless 0-RTT is also used; see Section 7.4.1 of [QUIC-TRANSPORT] and Section 4.6.1. Application protocols could depend on state that is retained between resumed connections.

Clients can store any state required for resumption along with the session ticket. Servers can use the session ticket to help carry state.

Session resumption allows servers to link activity on the original connection with the resumed connection, which might be a privacy issue for clients. Clients can choose not to enable resumption to avoid creating this correlation. Clients SHOULD NOT reuse tickets as that allows entities other than the server to correlate connections; see Section C.4 of [TLS13].

4.6. 0-RTT

The 0-RTT feature in QUIC allows a client to send application data before the handshake is complete. This is made possible by reusing negotiated parameters from a previous connection. To enable this, 0-RTT depends on the client remembering critical parameters and providing the server with a TLS session ticket that allows the server to recover the same information.

This information includes parameters that determine TLS state, as governed by [TLS13], QUIC transport parameters, the chosen application protocol, and any information the application protocol might need; see Section 4.6.3. This information determines how 0-RTT packets and their contents are formed.

To ensure that the same information is available to both endpoints, all information used to establish 0-RTT comes from the same connection. Endpoints cannot selectively disregard information that might alter the sending or processing of 0-RTT.

[TLS13] sets a limit of 7 days on the time between the original connection and any attempt to use 0-RTT. There are other constraints on 0-RTT usage, notably those caused by the potential exposure to replay attack; see Section 9.2.

4.6.1. Enabling 0-RTT

The TLS "early_data" extension in the NewSessionTicket message is defined to convey (in the "max_early_data_size" parameter) the amount of TLS 0-RTT data the server is willing to accept. QUIC does not use TLS 0-RTT data. QUIC uses 0-RTT packets to carry early data. Accordingly, the "max_early_data_size" parameter is repurposed to hold a sentinel value 0xffffffff to indicate that the server is willing to accept QUIC 0-RTT data; to indicate that the server does not accept 0-RTT data, the "early_data" extension is omitted from the NewSessionTicket. The amount of data that the client can send in QUIC 0-RTT is controlled by the initial_max_data transport parameter supplied by the server.

Servers MUST NOT send the early_data extension with a max_early_data_size field set to any value other than 0xffffffff. A client MUST treat receipt of a NewSessionTicket that contains an early_data extension with any other value as a connection error of type PROTOCOL_VIOLATION.

A client that wishes to send 0-RTT packets uses the `early_data` extension in the `ClientHello` message of a subsequent handshake; see Section 4.2.10 of [TLS13]. It then sends application data in 0-RTT packets.

A client that attempts 0-RTT might also provide an address validation token if the server has sent a `NEW_TOKEN` frame; see Section 8.1 of [QUIC-TRANSPORT].

4.6.2. Accepting and Rejecting 0-RTT

A server accepts 0-RTT by sending an `early_data` extension in the `EncryptedExtensions`; see Section 4.2.10 of [TLS13]. The server then processes and acknowledges the 0-RTT packets that it receives.

A server rejects 0-RTT by sending the `EncryptedExtensions` without an `early_data` extension. A server will always reject 0-RTT if it sends a `TLS HelloRetryRequest`. When rejecting 0-RTT, a server **MUST NOT** process any 0-RTT packets, even if it could. When 0-RTT was rejected, a client **SHOULD** treat receipt of an acknowledgment for a 0-RTT packet as a connection error of type `PROTOCOL_VIOLATION`, if it is able to detect the condition.

When 0-RTT is rejected, all connection characteristics that the client assumed might be incorrect. This includes the choice of application protocol, transport parameters, and any application configuration. The client therefore **MUST** reset the state of all streams, including application state bound to those streams.

A client **MAY** reattempt 0-RTT if it receives a `Retry` or `Version Negotiation` packet. These packets do not signify rejection of 0-RTT.

4.6.3. Validating 0-RTT Configuration

When a server receives a `ClientHello` with the `early_data` extension, it has to decide whether to accept or reject early data from the client. Some of this decision is made by the TLS stack (e.g., checking that the cipher suite being resumed was included in the `ClientHello`; see Section 4.2.10 of [TLS13]). Even when the TLS stack has no reason to reject early data, the QUIC stack or the application protocol using QUIC might reject early data because the configuration of the transport or application associated with the resumed session is not compatible with the server's current configuration.

QUIC requires additional transport state to be associated with a 0-RTT session ticket. One common way to implement this is using stateless session tickets and storing this state in the session ticket. Application protocols that use QUIC might have similar

requirements regarding associating or storing state. This associated state is used for deciding whether early data must be rejected. For example, HTTP/3 ([QUIC-HTTP]) settings determine how early data from the client is interpreted. Other applications using QUIC could have different requirements for determining whether to accept or reject early data.

4.7. HelloRetryRequest

The HelloRetryRequest message (see Section 4.1.4 of [TLS13]) can be used to request that a client provide new information, such as a key share, or to validate some characteristic of the client. From the perspective of QUIC, HelloRetryRequest is not differentiated from other cryptographic handshake messages that are carried in Initial packets. Although it is in principle possible to use this feature for address verification, QUIC implementations SHOULD instead use the Retry feature; see Section 8.1 of [QUIC-TRANSPORT].

4.8. TLS Errors

If TLS experiences an error, it generates an appropriate alert as defined in Section 6 of [TLS13].

A TLS alert is converted into a QUIC connection error. The AlertDescription value is added to 0x100 to produce a QUIC error code from the range reserved for CRYPTO_ERROR. The resulting value is sent in a QUIC CONNECTION_CLOSE frame of type 0x1c.

QUIC is only able to convey an alert level of "fatal". In TLS 1.3, the only existing uses for the "warning" level are to signal connection close; see Section 6.1 of [TLS13]. As QUIC provides alternative mechanisms for connection termination and the TLS connection is only closed if an error is encountered, a QUIC endpoint MUST treat any alert from TLS as if it were at the "fatal" level.

QUIC permits the use of a generic code in place of a specific error code; see Section 11 of [QUIC-TRANSPORT]. For TLS alerts, this includes replacing any alert with a generic alert, such as handshake_failure (0x128 in QUIC). Endpoints MAY use a generic error code to avoid possibly exposing confidential information.

4.9. Discarding Unused Keys

After QUIC has completed a move to a new encryption level, packet protection keys for previous encryption levels can be discarded. This occurs several times during the handshake, as well as when keys are updated; see Section 6.

Packet protection keys are not discarded immediately when new keys are available. If packets from a lower encryption level contain CRYPTO frames, frames that retransmit that data MUST be sent at the same encryption level. Similarly, an endpoint generates acknowledgments for packets at the same encryption level as the packet being acknowledged. Thus, it is possible that keys for a lower encryption level are needed for a short time after keys for a newer encryption level are available.

An endpoint cannot discard keys for a given encryption level unless it has received all the cryptographic handshake messages from its peer at that encryption level and its peer has done the same. Different methods for determining this are provided for Initial keys (Section 4.9.1) and Handshake keys (Section 4.9.2). These methods do not prevent packets from being received or sent at that encryption level because a peer might not have received all the acknowledgments necessary.

Though an endpoint might retain older keys, new data MUST be sent at the highest currently-available encryption level. Only ACK frames and retransmissions of data in CRYPTO frames are sent at a previous encryption level. These packets MAY also include PADDING frames.

4.9.1. Discarding Initial Keys

Packets protected with Initial secrets (Section 5.2) are not authenticated, meaning that an attacker could spoof packets with the intent to disrupt a connection. To limit these attacks, Initial packet protection keys are discarded more aggressively than other keys.

The successful use of Handshake packets indicates that no more Initial packets need to be exchanged, as these keys can only be produced after receiving all CRYPTO frames from Initial packets. Thus, a client MUST discard Initial keys when it first sends a Handshake packet and a server MUST discard Initial keys when it first successfully processes a Handshake packet. Endpoints MUST NOT send Initial packets after this point.

This results in abandoning loss recovery state for the Initial encryption level and ignoring any outstanding Initial packets.

4.9.2. Discarding Handshake Keys

An endpoint MUST discard its handshake keys when the TLS handshake is confirmed (Section 4.1.2).

4.9.3. Discarding 0-RTT Keys

0-RTT and 1-RTT packets share the same packet number space, and clients do not send 0-RTT packets after sending a 1-RTT packet (Section 5.6).

Therefore, a client SHOULD discard 0-RTT keys as soon as it installs 1-RTT keys, since they have no use after that moment.

Additionally, a server MAY discard 0-RTT keys as soon as it receives a 1-RTT packet. However, due to packet reordering, a 0-RTT packet could arrive after a 1-RTT packet. Servers MAY temporarily retain 0-RTT keys to allow decrypting reordered packets without requiring their contents to be retransmitted with 1-RTT keys. After receiving a 1-RTT packet, servers MUST discard 0-RTT keys within a short time; the RECOMMENDED time period is three times the Probe Timeout (PTO, see [QUIC-RECOVERY]). A server MAY discard 0-RTT keys earlier if it determines that it has received all 0-RTT packets, which can be done by keeping track of missing packet numbers.

5. Packet Protection

As with TLS over TCP, QUIC protects packets with keys derived from the TLS handshake, using the AEAD algorithm [AEAD] negotiated by TLS.

QUIC packets have varying protections depending on their type:

- * Version Negotiation packets have no cryptographic protection.
- * Retry packets use AEAD_AES_128_GCM to provide protection against accidental modification and to limit the entities that can produce a valid Retry; see Section 5.8.
- * Initial packets use AEAD_AES_128_GCM with keys derived from the Destination Connection ID field of the first Initial packet sent by the client; see Section 5.2.
- * All other packets have strong cryptographic protections for confidentiality and integrity, using keys and algorithms negotiated by TLS.

This section describes how packet protection is applied to Handshake packets, 0-RTT packets, and 1-RTT packets. The same packet protection process is applied to Initial packets. However, as it is trivial to determine the keys used for Initial packets, these packets are not considered to have confidentiality or integrity protection. Retry packets use a fixed key and so similarly lack confidentiality and integrity protection.

5.1. Packet Protection Keys

QUIC derives packet protection keys in the same way that TLS derives record protection keys.

Each encryption level has separate secret values for protection of packets sent in each direction. These traffic secrets are derived by TLS (see Section 7.1 of [TLS13]) and are used by QUIC for all encryption levels except the Initial encryption level. The secrets for the Initial encryption level are computed based on the client's initial Destination Connection ID, as described in Section 5.2.

The keys used for packet protection are computed from the TLS secrets using the KDF provided by TLS. In TLS 1.3, the HKDF-Expand-Label function described in Section 7.1 of [TLS13] is used, using the hash function from the negotiated cipher suite. All uses of HKDF-Expand-Label in QUIC use a zero-length Context.

Note that labels, which are described using strings, are encoded as bytes using ASCII [ASCII] without quotes or any trailing NUL byte.

Other versions of TLS MUST provide a similar function in order to be used with QUIC.

The current encryption level secret and the label "quic key" are input to the KDF to produce the AEAD key; the label "quic iv" is used to derive the Initialization Vector (IV); see Section 5.3. The header protection key uses the "quic hp" label; see Section 5.4. Using these labels provides key separation between QUIC and TLS; see Section 9.6.

Both "quic key" and "quic hp" are used to produce keys, so the Length provided to HKDF-Expand-Label along with these labels is determined by the size of keys in the AEAD or header protection algorithm. The Length provided with "quic iv" is the minimum length of the AEAD nonce, or 8 bytes if that is larger; see [AEAD].

The KDF used for initial secrets is always the HKDF-Expand-Label function from TLS 1.3; see Section 5.2.

5.2. Initial Secrets

Initial packets apply the packet protection process, but use a secret derived from the Destination Connection ID field from the client's first Initial packet.

This secret is determined by using HKDF-Extract (see Section 2.2 of [HKDF]) with a salt of 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a and a IKM of the Destination Connection ID field. This produces an intermediate pseudorandom key (PRK) that is used to derive two separate secrets for sending and receiving.

The secret used by clients to construct Initial packets uses the PRK and the label "client in" as input to the HKDF-Expand-Label function from TLS [TLS13] to produce a 32-byte secret. Packets constructed by the server use the same process with the label "server in". The hash function for HKDF when deriving initial secrets and keys is SHA-256 [SHA].

This process in pseudocode is:

```
initial_salt = 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a
initial_secret = HKDF-Extract(initial_salt,
                              client_dst_connection_id)

client_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "client in", "",
                                          Hash.length)
server_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "server in", "",
                                          Hash.length)
```

The connection ID used with HKDF-Expand-Label is the Destination Connection ID in the Initial packet sent by the client. This will be a randomly-selected value unless the client creates the Initial packet after receiving a Retry packet, where the Destination Connection ID is selected by the server.

Future versions of QUIC SHOULD generate a new salt value, thus ensuring that the keys are different for each version of QUIC. This prevents a middlebox that recognizes only one version of QUIC from seeing or modifying the contents of packets from future versions.

The HKDF-Expand-Label function defined in TLS 1.3 MUST be used for Initial packets even where the TLS versions offered do not include TLS 1.3.

The secrets used for constructing subsequent Initial packets change when a server sends a Retry packet, to use the connection ID value selected by the server. The secrets do not change when a client changes the Destination Connection ID it uses in response to an Initial packet from the server.

Note: The Destination Connection ID field could be any length up to

20 bytes, including zero length if the server sends a Retry packet with a zero-length Source Connection ID field. After a Retry, the Initial keys provide the client no assurance that the server received its packet, so the client has to rely on the exchange that included the Retry packet to validate the server address; see Section 8.1 of [QUIC-TRANSPORT].

Appendix A contains sample Initial packets.

5.3. AEAD Usage

The Authenticated Encryption with Associated Data (AEAD; see [AEAD]) function used for QUIC packet protection is the AEAD that is negotiated for use with the TLS connection. For example, if TLS is using the TLS_AES_128_GCM_SHA256 cipher suite, the AEAD_AES_128_GCM function is used.

QUIC can use any of the cipher suites defined in [TLS13] with the exception of TLS_AES_128_CCM_8_SHA256. A cipher suite MUST NOT be negotiated unless a header protection scheme is defined for the cipher suite. This document defines a header protection scheme for all cipher suites defined in [TLS13] aside from TLS_AES_128_CCM_8_SHA256. These cipher suites have a 16-byte authentication tag and produce an output 16 bytes larger than their input.

Note: An endpoint MUST NOT reject a ClientHello that offers a cipher suite that it does not support, or it would be impossible to deploy a new cipher suite. This also applies to TLS_AES_128_CCM_8_SHA256.

When constructing packets, the AEAD function is applied prior to applying header protection; see Section 5.4. The unprotected packet header is part of the associated data (A). When processing packets, an endpoint first removes the header protection.

The key and IV for the packet are computed as described in Section 5.1. The nonce, N, is formed by combining the packet protection IV with the packet number. The 62 bits of the reconstructed QUIC packet number in network byte order are left-padded with zeros to the size of the IV. The exclusive OR of the padded packet number and the IV forms the AEAD nonce.

The associated data, A, for the AEAD is the contents of the QUIC header, starting from the first byte of either the short or long header, up to and including the unprotected packet number.

The input plaintext, *P*, for the AEAD is the payload of the QUIC packet, as described in [QUIC-TRANSPORT].

The output ciphertext, *C*, of the AEAD is transmitted in place of *P*.

Some AEAD functions have limits for how many packets can be encrypted under the same key and IV; see Section 6.6. This might be lower than the packet number limit. An endpoint **MUST** initiate a key update (Section 6) prior to exceeding any limit set for the AEAD that is in use.

5.4. Header Protection

Parts of QUIC packet headers, in particular the Packet Number field, are protected using a key that is derived separately from the packet protection key and IV. The key derived using the "quic hp" label is used to provide confidentiality protection for those fields that are not exposed to on-path elements.

This protection applies to the least-significant bits of the first byte, plus the Packet Number field. The four least-significant bits of the first byte are protected for packets with long headers; the five least significant bits of the first byte are protected for packets with short headers. For both header forms, this covers the reserved bits and the Packet Number Length field; the Key Phase bit is also protected for packets with a short header.

The same header protection key is used for the duration of the connection, with the value not changing after a key update (see Section 6). This allows header protection to be used to protect the key phase.

This process does not apply to Retry or Version Negotiation packets, which do not contain a protected payload or any of the fields that are protected by this process.

5.4.1. Header Protection Application

Header protection is applied after packet protection is applied (see Section 5.3). The ciphertext of the packet is sampled and used as input to an encryption algorithm. The algorithm used depends on the negotiated AEAD.

The output of this algorithm is a 5-byte mask that is applied to the protected header fields using exclusive OR. The least significant bits of the first byte of the packet are masked by the least significant bits of the first mask byte, and the packet number is masked with the remaining bytes. Any unused bytes of mask that might result from a shorter packet number encoding are unused.

Figure 6 shows a sample algorithm for applying header protection. Removing header protection only differs in the order in which the packet number length (pn_length) is determined (here "^" is used to represent exclusive or).

```
mask = header_protection(hp_key, sample)

pn_length = (packet[0] & 0x03) + 1
if (packet[0] & 0x80) == 0x80:
    # Long header: 4 bits masked
    packet[0] ^= mask[0] & 0x0f
else:
    # Short header: 5 bits masked
    packet[0] ^= mask[0] & 0x1f

# pn_offset is the start of the Packet Number field.
packet[pn_offset:pn_offset+pn_length] ^= mask[1:1+pn_length]
```

Figure 6: Header Protection Pseudocode

Specific header protection functions are defined based on the selected cipher suite; see Section 5.4.3 and Section 5.4.4.

Figure 7 shows an example long header packet (Initial) and a short header packet (1-RTT). Figure 7 shows the fields in each header that are covered by header protection and the portion of the protected packet payload that is sampled.


```

Initial Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 0,
  Reserved Bits (2),      # Protected
  Packet Number Length (2), # Protected
  Version (32),
  DCID Len (8),
  Destination Connection ID (0..160),
  SCID Len (8),
  Source Connection ID (0..160),
  Token Length (i),
  Token (...),
  Length (i),
  Packet Number (8..32),      # Protected
  Protected Payload (0..24), # Skipped Part
  Protected Payload (128),    # Sampled Part
  Protected Payload (...)     # Remainder
}

1-RTT Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),      # Protected
  Key Phase (1),          # Protected
  Packet Number Length (2), # Protected
  Destination Connection ID (0..160),
  Packet Number (8..32),  # Protected
  Protected Payload (0..24), # Skipped Part
  Protected Payload (128), # Sampled Part
  Protected Payload (...), # Remainder
}

```

Figure 7: Header Protection and Ciphertext Sample

Before a TLS cipher suite can be used with QUIC, a header protection algorithm MUST be specified for the AEAD used with that cipher suite. This document defines algorithms for AEAD_AES_128_GCM, AEAD_AES_128_CCM, AEAD_AES_256_GCM (all these AES AEADs are defined in [AEAD]), and AEAD_CHACHA20_POLY1305 (defined in [CHACHA]). Prior to TLS selecting a cipher suite, AES header protection is used (Section 5.4.3), matching the AEAD_AES_128_GCM packet protection.

5.4.2. Header Protection Sample

The header protection algorithm uses both the header protection key and a sample of the ciphertext from the packet Payload field.

The same number of bytes are always sampled, but an allowance needs to be made for the endpoint removing protection, which will not know the length of the Packet Number field. The sample of ciphertext is taken starting from an offset of 4 bytes after the start of the Packet Number field. That is, in sampling packet ciphertext for header protection, the Packet Number field is assumed to be 4 bytes long (its maximum possible encoded length).

An endpoint **MUST** discard packets that are not long enough to contain a complete sample.

To ensure that sufficient data is available for sampling, packets are padded so that the combined lengths of the encoded packet number and protected payload is at least 4 bytes longer than the sample required for header protection. The cipher suites defined in [TLS13] - other than TLS_AES_128_GCM_SHA256, for which a header protection scheme is not defined in this document - have 16-byte expansions and 16-byte header protection samples. This results in needing at least 3 bytes of frames in the unprotected payload if the packet number is encoded on a single byte, or 2 bytes of frames for a 2-byte packet number encoding.

The sampled ciphertext can be determined by the following pseudocode:

```
# pn_offset is the start of the Packet Number field.
```

```
sample_offset = pn_offset + 4
```

```
sample = packet[sample_offset..sample_offset+sample_length]
```

where the packet number offset of a short header packet can be calculated as:

```
pn_offset = 1 + len(connection_id)
```

and the packet number offset of a long header packet can be calculated as:

```
pn_offset = 7 + len(destination_connection_id) +  
              len(source_connection_id) +  
              len(payload_length)
```

```
if packet_type == Initial:  
    pn_offset += len(token_length) +  
                len(token)
```

For example, for a packet with a short header, an 8-byte connection ID, and protected with AEAD_AES_128_GCM, the sample takes bytes 13 to 28 inclusive (using zero-based indexing).

Multiple QUIC packets might be included in the same UDP datagram. Each packet is handled separately.

5.4.3. AES-Based Header Protection

This section defines the packet protection algorithm for AEAD_AES_128_GCM, AEAD_AES_128_CCM, and AEAD_AES_256_GCM. AEAD_AES_128_GCM and AEAD_AES_128_CCM use 128-bit AES in electronic code-book (ECB) mode. AEAD_AES_256_GCM uses 256-bit AES in ECB mode. AES is defined in [AES].

This algorithm samples 16 bytes from the packet ciphertext. This value is used as the input to AES-ECB. In pseudocode, the header protection function is defined as:

```
header_protection(hp_key, sample):  
    mask = AES-ECB(hp_key, sample)
```

5.4.4. ChaCha20-Based Header Protection

When AEAD_CHACHA20_POLY1305 is in use, header protection uses the raw ChaCha20 function as defined in Section 2.4 of [CHACHA]. This uses a 256-bit key and 16 bytes sampled from the packet protection output.

The first 4 bytes of the sampled ciphertext are the block counter. A ChaCha20 implementation could take a 32-bit integer in place of a byte sequence, in which case the byte sequence is interpreted as a little-endian value.

The remaining 12 bytes are used as the nonce. A ChaCha20 implementation might take an array of three 32-bit integers in place of a byte sequence, in which case the nonce bytes are interpreted as a sequence of 32-bit little-endian integers.

The encryption mask is produced by invoking ChaCha20 to protect 5 zero bytes. In pseudocode, the header protection function is defined as:

```
header_protection(hp_key, sample):  
    counter = sample[0..3]  
    nonce = sample[4..15]  
    mask = ChaCha20(hp_key, counter, nonce, {0,0,0,0,0})
```

5.5. Receiving Protected Packets

Once an endpoint successfully receives a packet with a given packet number, it **MUST** discard all packets in the same packet number space with higher packet numbers if they cannot be successfully unprotected with either the same key, or - if there is a key update - a subsequent packet protection key; see Section 6. Similarly, a packet that appears to trigger a key update, but cannot be unprotected successfully **MUST** be discarded.

Failure to unprotect a packet does not necessarily indicate the existence of a protocol error in a peer or an attack. The truncated packet number encoding used in QUIC can cause packet numbers to be decoded incorrectly if they are delayed significantly.

5.6. Use of 0-RTT Keys

If 0-RTT keys are available (see Section 4.6.1), the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

Of the frames defined in [QUIC-TRANSPORT], the STREAM, RESET_STREAM, STOP_SENDING, and CONNECTION_CLOSE frames are potentially unsafe for use with 0-RTT as they carry application data. Application data that is received in 0-RTT could cause an application at the server to process the data multiple times rather than just once. Additional actions taken by a server as a result of processing replayed application data could have unwanted consequences. A client therefore **MUST NOT** use 0-RTT for application data unless specifically requested by the application that is in use.

An application protocol that uses QUIC **MUST** include a profile that defines acceptable use of 0-RTT; otherwise, 0-RTT can only be used to carry QUIC frames that do not carry application data. For example, a profile for HTTP is described in [HTTP-REPLAY] and used for HTTP/3; see Section 10.9 of [QUIC-HTTP].

Though replaying packets might result in additional connection attempts, the effect of processing replayed frames that do not carry application data is limited to changing the state of the affected connection. A TLS handshake cannot be successfully completed using replayed packets.

A client **MAY** wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake.

A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys, except that it cannot send certain frames with 0-RTT keys; see Section 12.5 of [QUIC-TRANSPORT].

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets; it uses 1-RTT keys to protect acknowledgments of 0-RTT packets. A client MUST NOT attempt to decrypt 0-RTT packets it receives and instead MUST discard them.

Once a client has installed 1-RTT keys, it MUST NOT send any more 0-RTT packets.

Note: 0-RTT data can be acknowledged by the server as it receives it, but any packets containing acknowledgments of 0-RTT data cannot have packet protection removed by the client until the TLS handshake is complete. The 1-RTT keys necessary to remove packet protection cannot be derived until the client receives all server handshake messages.

5.7. Receiving Out-of-Order Protected Packets

Due to reordering and loss, protected packets might be received by an endpoint before the final TLS handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client. Endpoints in either role MUST NOT decrypt 1-RTT packets from their peer prior to completing the handshake.

Even though 1-RTT keys are available to a server after receiving the first handshake messages from a client, it is missing assurances on the client state:

- * The client is not authenticated, unless the server has chosen to use a pre-shared key and validated the client's pre-shared key binder; see Section 4.2.11 of [TLS13].
- * The client has not demonstrated liveness, unless the server has validated the client's address with a Retry packet or other means; see Section 8.1 of [QUIC-TRANSPORT].
- * Any received 0-RTT data that the server responds to might be due to a replay attack.

Therefore, the server's use of 1-RTT keys before the handshake is complete is limited to sending data. A server MUST NOT process incoming 1-RTT protected packets before the TLS handshake is complete. Because sending acknowledgments indicates that all frames in a packet have been processed, a server cannot send acknowledgments for 1-RTT packets until the TLS handshake is complete. Received packets protected with 1-RTT keys MAY be stored and later decrypted and used once the handshake is complete.

Note: TLS implementations might provide all 1-RTT secrets prior to handshake completion. Even where QUIC implementations have 1-RTT read keys, those keys are not to be used prior to completing the handshake.

The requirement for the server to wait for the client Finished message creates a dependency on that message being delivered. A client can avoid the potential for head-of-line blocking that this implies by sending its 1-RTT packets coalesced with a Handshake packet containing a copy of the CRYPTO frame that carries the Finished message, until one of the Handshake packets is acknowledged. This enables immediate server processing for those packets.

A server could receive packets protected with 0-RTT keys prior to receiving a TLS ClientHello. The server MAY retain these packets for later decryption in anticipation of receiving a ClientHello.

A client generally receives 1-RTT keys at the same time as the handshake completes. Even if it has 1-RTT secrets, a client MUST NOT process incoming 1-RTT protected packets before the TLS handshake is complete.

5.8. Retry Packet Integrity

Retry packets (see the Retry Packet section of [QUIC-TRANSPORT]) carry a Retry Integrity Tag that provides two properties: it allows discarding packets that have accidentally been corrupted by the network; only an entity that observes an Initial packet can send a valid Retry packet.

The Retry Integrity Tag is a 128-bit field that is computed as the output of AEAD_AES_128_GCM ([AEAD]) used with the following inputs:

- * The secret key, K, is 128 bits equal to 0xbe0c690b9f66575a1d766b54e368c84e.
- * The nonce, N, is 96 bits equal to 0x461599d35d632bf2239825bb.
- * The plaintext, P, is empty.

- * The associated data, A, is the contents of the Retry Pseudo-Packet, as illustrated in Figure 8:

The secret key and the nonce are values derived by calling HKDF-Expand-Label using 0xd9c9943e6101fd200021506bcc02814c73030f25c79d71ce876eca876e6fca8e as the secret, with labels being "quic key" and "quic iv" (Section 5.1).

```
Retry Pseudo-Packet {  
  ODCID Length (8),  
  Original Destination Connection ID (0..160),  
  Header Form (1) = 1,  
  Fixed Bit (1) = 1,  
  Long Packet Type (2) = 3,  
  Unused (4),  
  Version (32),  
  DCID Len (8),  
  Destination Connection ID (0..160),  
  SCID Len (8),  
  Source Connection ID (0..160),  
  Retry Token (...),  
}
```

Figure 8: Retry Pseudo-Packet

The Retry Pseudo-Packet is not sent over the wire. It is computed by taking the transmitted Retry packet, removing the Retry Integrity Tag and prepending the two following fields:

ODCID Length: The ODCID Length field contains the length in bytes of the Original Destination Connection ID field that follows it, encoded as an 8-bit unsigned integer.

Original Destination Connection ID: The Original Destination Connection ID contains the value of the Destination Connection ID from the Initial packet that this Retry is in response to. The length of this field is given in ODCID Length. The presence of this field ensures that a valid Retry packet can only be sent by an entity that observes the Initial packet.

6. Key Update

Once the handshake is confirmed (see Section 4.1.2), an endpoint MAY initiate a key update.

The Key Phase bit indicates which packet protection keys are used to protect the packet. The Key Phase bit is initially set to 0 for the first set of 1-RTT packets and toggled to signal each subsequent key update.

The Key Phase bit allows a recipient to detect a change in keying material without needing to receive the first packet that triggered the change. An endpoint that notices a changed Key Phase bit updates keys and decrypts the packet that contains the changed value.

Initiating a key update results in both endpoints updating keys. This differs from TLS where endpoints can update keys independently.

This mechanism replaces the key update mechanism of TLS, which relies on KeyUpdate messages sent using 1-RTT encryption keys. Endpoints **MUST NOT** send a TLS KeyUpdate message. Endpoints **MUST** treat the receipt of a TLS KeyUpdate message as a connection error of type 0x10a, equivalent to a fatal TLS alert of `unexpected_message`; see Section 4.8.

Figure 9 shows a key update process, where the initial set of keys used (identified with @M) are replaced by updated keys (identified with @N). The value of the Key Phase bit is indicated in brackets [].

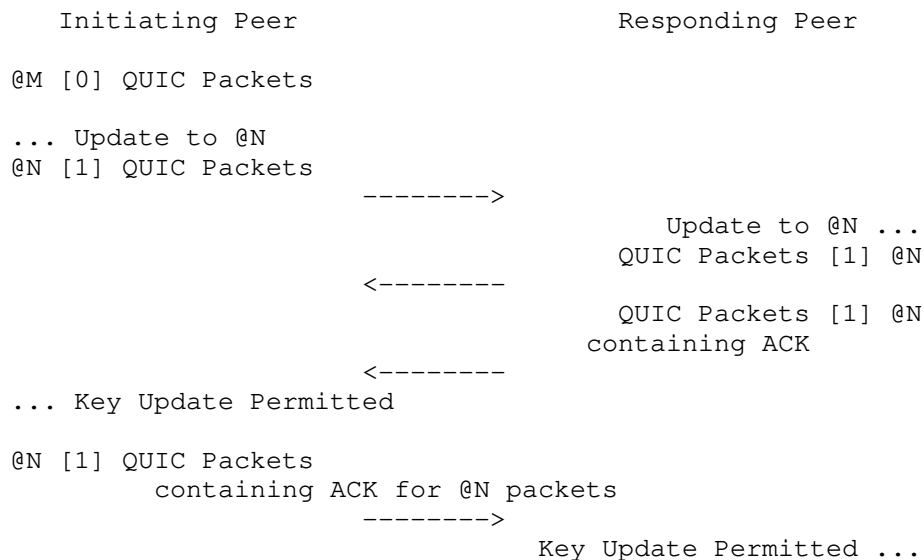


Figure 9: Key Update

6.1. Initiating a Key Update

Endpoints maintain separate read and write secrets for packet protection. An endpoint initiates a key update by updating its packet protection write secret and using that to protect new packets. The endpoint creates a new write secret from the existing write secret as performed in Section 7.2 of [TLS13]. This uses the KDF function provided by TLS with a label of "quic ku". The corresponding key and IV are created from that secret as defined in Section 5.1. The header protection key is not updated.

For example, to update write keys with TLS 1.3, HKDF-Expand-Label is used as:

```
secret_<n+1> = HKDF-Expand-Label(secret_<n>, "quic ku",  
                                "", Hash.length)
```

The endpoint toggles the value of the Key Phase bit and uses the updated key and IV to protect all subsequent packets.

An endpoint **MUST NOT** initiate a key update prior to having confirmed the handshake (Section 4.1.2). An endpoint **MUST NOT** initiate a subsequent key update unless it has received an acknowledgment for a packet that was sent protected with keys from the current key phase. This ensures that keys are available to both peers before another key update can be initiated. This can be implemented by tracking the lowest packet number sent with each key phase, and the highest acknowledged packet number in the 1-RTT space: once the latter is higher than or equal to the former, another key update can be initiated.

Note: Keys of packets other than the 1-RTT packets are never updated; their keys are derived solely from the TLS handshake state.

The endpoint that initiates a key update also updates the keys that it uses for receiving packets. These keys will be needed to process packets the peer sends after updating.

An endpoint **MUST** retain old keys until it has successfully unprotected a packet sent using the new keys. An endpoint **SHOULD** retain old keys for some time after unprotecting a packet sent using the new keys. Discarding old keys too early can cause delayed packets to be discarded. Discarding packets will be interpreted as packet loss by the peer and could adversely affect performance.

6.2. Responding to a Key Update

A peer is permitted to initiate a key update after receiving an acknowledgment of a packet in the current key phase. An endpoint detects a key update when processing a packet with a key phase that differs from the value used to protect the last packet it sent. To process this packet, the endpoint uses the next packet protection key and IV. See Section 6.3 for considerations about generating these keys.

If a packet is successfully processed using the next key and IV, then the peer has initiated a key update. The endpoint **MUST** update its send keys to the corresponding key phase in response, as described in Section 6.1. Sending keys **MUST** be updated before sending an acknowledgment for the packet that was received with updated keys. By acknowledging the packet that triggered the key update in a packet protected with the updated keys, the endpoint signals that the key update is complete.

An endpoint can defer sending the packet or acknowledgment according to its normal packet sending behaviour; it is not necessary to immediately generate a packet in response to a key update. The next packet sent by the endpoint will use the updated keys. The next packet that contains an acknowledgment will cause the key update to be completed. If an endpoint detects a second update before it has sent any packets with updated keys containing an acknowledgment for the packet that initiated the key update, it indicates that its peer has updated keys twice without awaiting confirmation. An endpoint **MAY** treat such consecutive key updates as a connection error of type `KEY_UPDATE_ERROR`.

An endpoint that receives an acknowledgment that is carried in a packet protected with old keys where any acknowledged packet was protected with newer keys **MAY** treat that as a connection error of type `KEY_UPDATE_ERROR`. This indicates that a peer has received and acknowledged a packet that initiates a key update, but has not updated keys in response.

6.3. Timing of Receive Key Generation

Endpoints responding to an apparent key update **MUST NOT** generate a timing side-channel signal that might indicate that the Key Phase bit was invalid (see Section 9.4). Endpoints can use dummy packet protection keys in place of discarded keys when key updates are not yet permitted. Using dummy keys will generate no variation in the timing signal produced by attempting to remove packet protection, and results in all packets with an invalid Key Phase bit being rejected.

The process of creating new packet protection keys for receiving packets could reveal that a key update has occurred. An endpoint MAY generate new keys as part of packet processing, but this creates a timing signal that could be used by an attacker to learn when key updates happen and thus leak the value of the Key Phase bit.

Endpoints are generally expected to have current and next receive packet protection keys available. For a short period after a key update completes, up to the PTO, endpoints MAY defer generation of the next set of receive packet protection keys. This allows endpoints to retain only two sets of receive keys; see Section 6.5.

Once generated, the next set of packet protection keys SHOULD be retained, even if the packet that was received was subsequently discarded. Packets containing apparent key updates are easy to forge and - while the process of key update does not require significant effort - triggering this process could be used by an attacker for DoS.

For this reason, endpoints MUST be able to retain two sets of packet protection keys for receiving packets: the current and the next. Retaining the previous keys in addition to these might improve performance, but this is not essential.

6.4. Sending with Updated Keys

An endpoint never sends packets that are protected with old keys. Only the current keys are used. Keys used for protecting packets can be discarded immediately after switching to newer keys.

Packets with higher packet numbers MUST be protected with either the same or newer packet protection keys than packets with lower packet numbers. An endpoint that successfully removes protection with old keys when newer keys were used for packets with lower packet numbers MUST treat this as a connection error of type KEY_UPDATE_ERROR.

6.5. Receiving with Different Keys

For receiving packets during a key update, packets protected with older keys might arrive if they were delayed by the network. Retaining old packet protection keys allows these packets to be successfully processed.

As packets protected with keys from the next key phase use the same Key Phase value as those protected with keys from the previous key phase, it is necessary to distinguish between the two, if packets protected with old keys are to be processed. This can be done using packet numbers. A recovered packet number that is lower than any

packet number from the current key phase uses the previous packet protection keys; a recovered packet number that is higher than any packet number from the current key phase requires the use of the next packet protection keys.

Some care is necessary to ensure that any process for selecting between previous, current, and next packet protection keys does not expose a timing side channel that might reveal which keys were used to remove packet protection. See Section 9.5 for more information.

Alternatively, endpoints can retain only two sets of packet protection keys, swapping previous for next after enough time has passed to allow for reordering in the network. In this case, the Key Phase bit alone can be used to select keys.

An endpoint MAY allow a period of approximately the Probe Timeout (PTO; see [QUIC-RECOVERY]) after promoting the next set of receive keys to be current before it creates the subsequent set of packet protection keys. These updated keys MAY replace the previous keys at that time. With the caveat that PTO is a subjective measure – that is, a peer could have a different view of the RTT – this time is expected to be long enough that any reordered packets would be declared lost by a peer even if they were acknowledged and short enough to allow a peer to initiate further key updates.

Endpoints need to allow for the possibility that a peer might not be able to decrypt packets that initiate a key update during the period when the peer retains old keys. Endpoints SHOULD wait three times the PTO before initiating a key update after receiving an acknowledgment that confirms that the previous key update was received. Failing to allow sufficient time could lead to packets being discarded.

An endpoint SHOULD retain old read keys for no more than three times the PTO after having received a packet protected using the new keys. After this period, old read keys and their corresponding secrets SHOULD be discarded.

6.6. Limits on AEAD Usage

This document sets usage limits for AEAD algorithms to ensure that overuse does not give an adversary a disproportionate advantage in attacking the confidentiality and integrity of communications when using QUIC.

The usage limits defined in TLS 1.3 exist for protection against attacks on confidentiality and apply to successful applications of AEAD protection. The integrity protections in authenticated

encryption also depend on limiting the number of attempts to forge packets. TLS achieves this by closing connections after any record fails an authentication check. In comparison, QUIC ignores any packet that cannot be authenticated, allowing multiple forgery attempts.

QUIC accounts for AEAD confidentiality and integrity limits separately. The confidentiality limit applies to the number of packets encrypted with a given key. The integrity limit applies to the number of packets decrypted within a given connection. Details on enforcing these limits for each AEAD algorithm follow below.

Endpoints MUST count the number of encrypted packets for each set of keys. If the total number of encrypted packets with the same key exceeds the confidentiality limit for the selected AEAD, the endpoint MUST stop using those keys. Endpoints MUST initiate a key update before sending more protected packets than the confidentiality limit for the selected AEAD permits. If a key update is not possible or integrity limits are reached, the endpoint MUST stop using the connection and only send stateless resets in response to receiving packets. It is RECOMMENDED that endpoints immediately close the connection with a connection error of type AEAD_LIMIT_REACHED before reaching a state where key updates are not possible.

For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the confidentiality limit is 2^{23} encrypted packets; see Appendix B.1. For AEAD_CHACHA20_POLY1305, the confidentiality limit is greater than the number of possible packets (2^{62}) and so can be disregarded. For AEAD_AES_128_CCM, the confidentiality limit is $2^{21.5}$ encrypted packets; see Appendix B.2. Applying a limit reduces the probability that an attacker can distinguish the AEAD in use from a random permutation; see [AEBounds], [ROBUST], and [GCM-MU].

In addition to counting packets sent, endpoints MUST count the number of received packets that fail authentication during the lifetime of a connection. If the total number of received packets that fail authentication within the connection, across all keys, exceeds the integrity limit for the selected AEAD, the endpoint MUST immediately close the connection with a connection error of type AEAD_LIMIT_REACHED and not process any more packets.

For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the integrity limit is 2^{52} invalid packets; see Appendix B.1. For AEAD_CHACHA20_POLY1305, the integrity limit is 2^{36} invalid packets; see [AEBounds]. For AEAD_AES_128_CCM, the integrity limit is $2^{21.5}$ invalid packets; see Appendix B.2. Applying this limit reduces the probability that an attacker can successfully forge a packet; see [AEBounds], [ROBUST], and [GCM-MU].

Endpoints that limit the size of packets MAY use higher confidentiality and integrity limits; see Appendix B for details.

Future analyses and specifications MAY relax confidentiality or integrity limits for an AEAD.

Any TLS cipher suite that is specified for use with QUIC MUST define limits on the use of the associated AEAD function that preserves margins for confidentiality and integrity. That is, limits MUST be specified for the number of packets that can be authenticated and for the number of packets that can fail authentication. Providing a reference to any analysis upon which values are based - and any assumptions used in that analysis - allows limits to be adapted to varying usage conditions.

6.7. Key Update Error Code

The KEY_UPDATE_ERROR error code (0xe) is used to signal errors related to key updates.

7. Security of Initial Messages

Initial packets are not protected with a secret key, so they are subject to potential tampering by an attacker. QUIC provides protection against attackers that cannot read packets, but does not attempt to provide additional protection against attacks where the attacker can observe and inject packets. Some forms of tampering -- such as modifying the TLS messages themselves -- are detectable, but some -- such as modifying ACKs -- are not.

For example, an attacker could inject a packet containing an ACK frame that makes it appear that a packet had not been received or to create a false impression of the state of the connection (e.g., by modifying the ACK Delay). Note that such a packet could cause a legitimate packet to be dropped as a duplicate. Implementations SHOULD use caution in relying on any data that is contained in Initial packets that is not otherwise authenticated.

It is also possible for the attacker to tamper with data that is carried in Handshake packets, but because that tampering requires modifying TLS handshake messages, that tampering will cause the TLS handshake to fail.

8. QUIC-Specific Adjustments to the TLS Handshake

Certain aspects of the TLS handshake are different when used with QUIC.

QUIC also requires additional features from TLS. In addition to negotiation of cryptographic parameters, the TLS handshake carries and authenticates values for QUIC transport parameters.

8.1. Protocol Negotiation

QUIC requires that the cryptographic handshake provide authenticated protocol negotiation. TLS uses Application Layer Protocol Negotiation ([ALPN]) to select an application protocol. Unless another mechanism is used for agreeing on an application protocol, endpoints MUST use ALPN for this purpose.

When using ALPN, endpoints MUST immediately close a connection (see Section 10.2 of [QUIC-TRANSPORT]) with a `no_application_protocol` TLS alert (QUIC error code 0x178; see Section 4.8) if an application protocol is not negotiated. While [ALPN] only specifies that servers use this alert, QUIC clients MUST use error 0x178 to terminate a connection when ALPN negotiation fails.

An application protocol MAY restrict the QUIC versions that it can operate over. Servers MUST select an application protocol compatible with the QUIC version that the client has selected. The server MUST treat the inability to select a compatible application protocol as a connection error of type 0x178 (`no_application_protocol`). Similarly, a client MUST treat the selection of an incompatible application protocol by a server as a connection error of type 0x178.

8.2. QUIC Transport Parameters Extension

QUIC transport parameters are carried in a TLS extension. Different versions of QUIC might define a different method for negotiating transport configuration.

Including transport parameters in the TLS handshake provides integrity protection for these values.

```
enum {  
    quic_transport_parameters(0x39), (65535)  
} ExtensionType;
```

The `extension_data` field of the `quic_transport_parameters` extension contains a value that is defined by the version of QUIC that is in use.

The `quic_transport_parameters` extension is carried in the `ClientHello` and the `EncryptedExtensions` messages during the handshake. Endpoints MUST send the `quic_transport_parameters` extension; endpoints that receive `ClientHello` or `EncryptedExtensions` messages without the

quic_transport_parameters extension MUST close the connection with an error of type 0x16d (equivalent to a fatal TLS missing_extension alert, see Section 4.8).

Transport parameters become available prior to the completion of the handshake. A server might use these values earlier than handshake completion. However, the value of transport parameters is not authenticated until the handshake completes, so any use of these parameters cannot depend on their authenticity. Any tampering with transport parameters will cause the handshake to fail.

Endpoints MUST NOT send this extension in a TLS connection that does not use QUIC (such as the use of TLS with TCP defined in [TLS13]). A fatal unsupported_extension alert MUST be sent by an implementation that supports this extension if the extension is received when the transport is not QUIC.

Negotiating the quic_transport_parameters extension causes the EndOfEarlyData to be removed; see Section 8.3.

8.3. Removing the EndOfEarlyData Message

The TLS EndOfEarlyData message is not used with QUIC. QUIC does not rely on this message to mark the end of 0-RTT data or to signal the change to Handshake keys.

Clients MUST NOT send the EndOfEarlyData message. A server MUST treat receipt of a CRYPTO frame in a 0-RTT packet as a connection error of type PROTOCOL_VIOLATION.

As a result, EndOfEarlyData does not appear in the TLS handshake transcript.

8.4. Prohibit TLS Middlebox Compatibility Mode

Appendix D.4 of [TLS13] describes an alteration to the TLS 1.3 handshake as a workaround for bugs in some middleboxes. The TLS 1.3 middlebox compatibility mode involves setting the legacy_session_id field to a 32-byte value in the ClientHello and ServerHello, then sending a change_cipher_spec record. Both field and record carry no semantic content and are ignored.

This mode has no use in QUIC as it only applies to middleboxes that interfere with TLS over TCP. QUIC also provides no means to carry a change_cipher_spec record. A client MUST NOT request the use of the TLS 1.3 compatibility mode. A server SHOULD treat the receipt of a TLS ClientHello with a non-empty legacy_session_id field as a connection error of type PROTOCOL_VIOLATION.

9. Security Considerations

All of the security considerations that apply to TLS also apply to the use of TLS in QUIC. Reading all of [TLS13] and its appendices is the best way to gain an understanding of the security properties of QUIC.

This section summarizes some of the more important security aspects specific to the TLS integration, though there are many security-relevant details in the remainder of the document.

9.1. Session Linkability

Use of TLS session tickets allows servers and possibly other entities to correlate connections made by the same client; see Section 4.5 for details.

9.2. Replay Attacks with 0-RTT

As described in Section 8 of [TLS13], use of TLS early data comes with an exposure to replay attack. The use of 0-RTT in QUIC is similarly vulnerable to replay attack.

Endpoints **MUST** implement and use the replay protections described in [TLS13], however it is recognized that these protections are imperfect. Therefore, additional consideration of the risk of replay is needed.

QUIC is not vulnerable to replay attack, except via the application protocol information it might carry. The management of QUIC protocol state based on the frame types defined in [QUIC-TRANSPORT] is not vulnerable to replay. Processing of QUIC frames is idempotent and cannot result in invalid connection states if frames are replayed, reordered or lost. QUIC connections do not produce effects that last beyond the lifetime of the connection, except for those produced by the application protocol that QUIC serves.

Note: TLS session tickets and address validation tokens are used to carry QUIC configuration information between connections. Specifically, to enable a server to efficiently recover state that is used in connection establishment and address validation. These **MUST NOT** be used to communicate application semantics between endpoints; clients **MUST** treat them as opaque values. The potential for reuse of these tokens means that they require stronger protections against replay.

A server that accepts 0-RTT on a connection incurs a higher cost than accepting a connection without 0-RTT. This includes higher processing and computation costs. Servers need to consider the probability of replay and all associated costs when accepting 0-RTT.

Ultimately, the responsibility for managing the risks of replay attacks with 0-RTT lies with an application protocol. An application protocol that uses QUIC MUST describe how the protocol uses 0-RTT and the measures that are employed to protect against replay attack. An analysis of replay risk needs to consider all QUIC protocol features that carry application semantics.

Disabling 0-RTT entirely is the most effective defense against replay attack.

QUIC extensions MUST describe how replay attacks affect their operation, or prohibit their use in 0-RTT. Application protocols MUST either prohibit the use of extensions that carry application semantics in 0-RTT or provide replay mitigation strategies.

9.3. Packet Reflection Attack Mitigation

A small ClientHello that results in a large block of handshake messages from a server can be used in packet reflection attacks to amplify the traffic generated by an attacker.

QUIC includes three defenses against this attack. First, the packet containing a ClientHello MUST be padded to a minimum size. Second, if responding to an unverified source address, the server is forbidden to send more than three times as many bytes as the number of bytes it has received (see Section 8.1 of [QUIC-TRANSPORT]). Finally, because acknowledgments of Handshake packets are authenticated, a blind attacker cannot forge them. Put together, these defenses limit the level of amplification.

9.4. Header Protection Analysis

[NAN] analyzes authenticated encryption algorithms that provide nonce privacy, referred to as "Hide Nonce" (HN) transforms. The general header protection construction in this document is one of those algorithms (HN1). Header protection is applied after the packet protection AEAD, sampling a set of bytes ("sample") from the AEAD output and encrypting the header field using a pseudorandom function (PRF) as follows:

```
protected_field = field XOR PRF(hp_key, sample)
```

The header protection variants in this document use a pseudorandom permutation (PRP) in place of a generic PRF. However, since all PRPs are also PRFs [IMC], these variants do not deviate from the HN1 construction.

As "hp_key" is distinct from the packet protection key, it follows that header protection achieves AE2 security as defined in [NAN] and therefore guarantees privacy of "field", the protected packet header. Future header protection variants based on this construction MUST use a PRF to ensure equivalent security guarantees.

Use of the same key and ciphertext sample more than once risks compromising header protection. Protecting two different headers with the same key and ciphertext sample reveals the exclusive OR of the protected fields. Assuming that the AEAD acts as a PRF, if L bits are sampled, the odds of two ciphertext samples being identical approach $2^{(-L/2)}$, that is, the birthday bound. For the algorithms described in this document, that probability is one in 2^{64} .

To prevent an attacker from modifying packet headers, the header is transitively authenticated using packet protection; the entire packet header is part of the authenticated additional data. Protected fields that are falsified or modified can only be detected once the packet protection is removed.

9.5. Header Protection Timing Side-Channels

An attacker could guess values for packet numbers or Key Phase and have an endpoint confirm guesses through timing side channels. Similarly, guesses for the packet number length can be tried and exposed. If the recipient of a packet discards packets with duplicate packet numbers without attempting to remove packet protection they could reveal through timing side-channels that the packet number matches a received packet. For authentication to be free from side-channels, the entire process of header protection removal, packet number recovery, and packet protection removal MUST be applied together without timing and other side-channels.

For the sending of packets, construction and protection of packet payloads and packet numbers MUST be free from side-channels that would reveal the packet number or its encoded size.

During a key update, the time taken to generate new keys could reveal through timing side-channels that a key update has occurred. Alternatively, where an attacker injects packets this side-channel could reveal the value of the Key Phase on injected packets. After receiving a key update, an endpoint SHOULD generate and save the next set of receive packet protection keys, as described in Section 6.3.

By generating new keys before a key update is received, receipt of packets will not create timing signals that leak the value of the Key Phase.

This depends on not doing this key generation during packet processing and it can require that endpoints maintain three sets of packet protection keys for receiving: for the previous key phase, for the current key phase, and for the next key phase. Endpoints can instead choose to defer generation of the next receive packet protection keys until they discard old keys so that only two sets of receive keys need to be retained at any point in time.

9.6. Key Diversity

In using TLS, the central key schedule of TLS is used. As a result of the TLS handshake messages being integrated into the calculation of secrets, the inclusion of the QUIC transport parameters extension ensures that handshake and 1-RTT keys are not the same as those that might be produced by a server running TLS over TCP. To avoid the possibility of cross-protocol key synchronization, additional measures are provided to improve key separation.

The QUIC packet protection keys and IVs are derived using a different label than the equivalent keys in TLS.

To preserve this separation, a new version of QUIC SHOULD define new labels for key derivation for packet protection key and IV, plus the header protection keys. This version of QUIC uses the string "quic". Other versions can use a version-specific label in place of that string.

The initial secrets use a key that is specific to the negotiated QUIC version. New QUIC versions SHOULD define a new salt value used in calculating initial secrets.

9.7. Randomness

QUIC depends on endpoints being able to generate secure random numbers, both directly for protocol values such as the connection ID, and transitively via TLS. See [RFC4086] for guidance on secure random number generation.

10. IANA Considerations

IANA has registered a codepoint of 57 (or 0x39) for the quic_transport_parameters extension (defined in Section 8.2) in the TLS ExtensionType Values Registry [TLS-REGISTRIES].

The Recommended column for this extension is marked Yes. The TLS 1.3 Column includes CH and EE.

11. References

11.1. Normative References

- [AEAD] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.
- [AES] "Advanced encryption standard (AES)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.197, November 2001, <<https://doi.org/10.6028/nist.fips.197>>.
- [ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [HKDF] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-34, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-recovery-34>>.
- [QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-transport-34>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4086>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [SHA] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015, <<https://doi.org/10.6028/nist.fips.180-4>>.
- [TLS-REGISTRIES] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <<https://www.rfc-editor.org/info/rfc8447>>.
- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

11.2. Informative References

- [AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 8 March 2016, <<http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>>.
- [ASCII] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <<https://www.rfc-editor.org/info/rfc20>>.
- [CCM-ANALYSIS] Jonsson, J., "On the Security of CTR + CBC-MAC", Selected Areas in Cryptography pp. 76-93, DOI 10.1007/3-540-36492-7_7, 2003, <https://doi.org/10.1007/3-540-36492-7_7>.
- [COMPRESS] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", Work in Progress, Internet-Draft, draft-ietf-tls-certificate-compression-10, 6 January 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-certificate-compression-10.txt>>.
- [GCM-MU] Hoang, V., Tessaro, S., and A. Thiruvengadam, "The Multi-user Security of GCM, Revisited: Tight Bounds for Nonce Randomization", Proceedings of the 2018 ACM SIGSAC

Conference on Computer and Communications Security,
DOI 10.1145/3243734.3243816, January 2018,
<<https://doi.org/10.1145/3243734.3243816>>.

[HTTP-REPLAY]

Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/info/rfc8470>>.

[HTTP2-TLS13]

Benjamin, D., "Using TLS 1.3 with HTTP/2", RFC 8740, DOI 10.17487/RFC8740, February 2020, <<https://www.rfc-editor.org/info/rfc8740>>.

[IMC]

Katz, J. and Y. Lindell, "Introduction to Modern Cryptography, Second Edition", ISBN 978-1466570269, 6 November 2014.

[NAN]

Bellare, M., Ng, R., and B. Tackmann, "Nonces Are Noticed: AEAD Revisited", Advances in Cryptology - CRYPTO 2019 pp. 235-265, DOI 10.1007/978-3-030-26948-7_9, 2019, <https://doi.org/10.1007/978-3-030-26948-7_9>.

[QUIC-HTTP]

Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-33, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-http-33>>.

[RFC2818]

Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/info/rfc2818>>.

[RFC5280]

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.

[ROBUST]

Fischlin, M., Günther, F., and C. Janson, "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3", 16 May 2020, <<https://eprint.iacr.org/2020/718>>.

Appendix A. Sample Packet Protection

This section shows examples of packet protection so that implementations can be verified incrementally. Samples of Initial packets from both client and server, plus a Retry packet are defined. These packets use an 8-byte client-chosen Destination Connection ID of 0x8394c8f03e515708. Some intermediate values are included. All values are shown in hexadecimal.

A.1. Keys

The labels generated during the execution of the HKDF-Expand-Label function (that is, `HkdfLabel.label`) and part of the value given to the HKDF-Expand function in order to produce its output are:

client in: 00200f746c73313320636c69656e7420696e00

server in: 00200f746c7331332073657276657220696e00

quic key: 00100e746c7331332071756963206b657900

quic iv: 000c0d746c733133207175696320697600

quic hp: 00100d746c733133207175696320687000

The initial secret is common:

```
initial_secret = HKDF-Extract(initial_salt, cid)
                = 7db5df06e7a69e432496adedb0085192
                  3595221596ae2ae9fb8115cle9ed0a44
```

The secrets for protecting client packets are:

```
client_initial_secret
    = HKDF-Expand-Label(initial_secret, "client in", "", 32)
    = c00cf151ca5be075ed0ebfb5c80323c4
      2d6b7db67881289af4008f1f6c357aea

key = HKDF-Expand-Label(client_initial_secret, "quic key", "", 16)
    = 1f369613dd76d5467730efcbe3b1a22d

iv  = HKDF-Expand-Label(client_initial_secret, "quic iv", "", 12)
    = fa044b2f42a3fd3b46fb255c

hp  = HKDF-Expand-Label(client_initial_secret, "quic hp", "", 16)
    = 9f50449e04a0e810283a1e9933adedd2
```

The secrets for protecting server packets are:


```

server_initial_secret
  = HKDF-Expand-Label(initial_secret, "server in", "", 32)
  = 3c199828fd139efd216c155ad844cc81
    fb82fa8d7446fa7d78be803acdda951b

key = HKDF-Expand-Label(server_initial_secret, "quic key", "", 16)
    = cf3a5331653c364c88f0f379b6067e37

iv  = HKDF-Expand-Label(server_initial_secret, "quic iv", "", 12)
    = 0ac1493ca1905853b0bba03e

hp  = HKDF-Expand-Label(server_initial_secret, "quic hp", "", 16)
    = c206b8d9b9f0f37644430b490eeaa314

```

A.2. Client Initial

The client sends an Initial packet. The unprotected payload of this packet contains the following CRYPTO frame, plus enough PADDING frames to make a 1162-byte payload:

```

060040f1010000ed0303ebf8fa56f129 39b9584a3896472ec40bb863cfd3e868
04fe3a47f06a2b69484c000004130113 02010000c000000010000e00000b6578
616d706c652e636f6dff01000100000a 00080006001d00170018001000070005
04616c706e0005000501000000000033 00260024001d00209370b2c9caa47fba
baf4559fedba753de171fa71f50f1ce1 5d43e994ec74d748002b000302030400
0d00100000e0403050306030203080408 050806002d00020101001c0002400100
3900320408fffffffffffffffff050480 00ffff07048000ffff08011001048000
75300901100f088394c8f03e51570806 048000ffff

```

The unprotected header indicates a length of 1182 bytes: the 4-byte packet number, 1162 bytes of frames, and the 16-byte authentication tag. The header includes the connection ID and a packet number of 2:

```
c300000001088394c8f03e5157080000449e00000002
```

Protecting the payload produces output that is sampled for header protection. Because the header uses a 4-byte packet number encoding, the first 16 bytes of the protected payload is sampled, then applied to the header:

```
sample = d1b1c98dd7689fb8ec11d242b123dc9b
mask = AES-ECB(hp, sample)[0..4]
      = 437b9aec36
header[0] ^= mask[0] & 0x0f
        = c0
header[18..21] ^= mask[1..4]
              = 7b9aec34
header = c000000001088394c8f03e5157080000449e7b9aec34
```

The resulting protected packet is:

```
c000000001088394c8f03e5157080000 449e7b9aec34dlb1c98dd7689fb8ec11
d242b123dc9bd8bab936b47d92ec356c 0bab7df5976d27cd449f63300099f399
1c260ec4c60d17b31f8429157bb35a12 82a643a8d2262cad67500cadb8e7378c
8eb7539ec4d4905fedlbee1fc8aafba1 7c750e2c7ace01e6005f80fcb7df6212
30c83711b39343fa028cea7f7fb5ff89 eac2308249a02252155e2347b63d58c5
457afd84d05dfffdb20392844ae81215 4682e9cf012f9021a6f0be17ddd0c208
4dce25ff9b06cde535d0f920a2db1bf3 62c23e596d11a4f5a6cf3948838a3aec
4e15daf8500a6ef69ec4e3feb6b1d98e 610ac8b7ec3faf6ad760b7bad1db4ba3
485e8a94dc250ae3fdb41ed15fb6a8e5 eba0fc3dd60bc8e30c5c4287e53805db
059ae0648db2f64264ed5e39be2e20d8 2df566da8dd5998ccabdae053060ae6c
7b4378e846d29f37ed7b4ea9ec5d82e7 961b7f25a9323851f681d582363aa5f8
9937f5a67258bf63ad6f1a0bld96dbd4 faddfcef5266ba6611722395c906556
be52afe3f565636ad1b17d508b73d874 3eeb524be22b3dcbc2c7468d54119c74
68449a13d8e3b95811a198f3491de3e7 fe942b330407abf82a4ed7c1b311663a
c69890f4157015853d91e923037c227a 33cdd5ec281ca3f79c44546b9d90ca00
f064c99e3dd97911d39fe9c5d0b23a22 9a234cb36186c4819e8b9c5927726632
291d6a418211cc2962e20fe47feb3edf 330f2c603a9d48c0fcb5699dbfe58964
25c5bac4aee82e57a85aaf4e2513e4f0 5796b07ba2ee47d80506f8d2c25e50fd
14de71e6c418559302f939b0e1abd576 f279c4b2e0feb85c1f28ff18f58891ff
ef132eef2fa09346aee33c28eb130ff2 8f5b766953334113211996d20011a198
e3fc433f9f2541010ae17c1bf202580f 6047472fb36857fe843b19f5984009dd
c324044e847a4f4a0ab34f719595de37 252d6235365e9b84392b061085349d73
203a4a13e96f5432ec0fd4alee65accd d5e3904df54c1da510b0ff20dcc0c77f
cb2c0e0eb605cb0504db87632cf3d8b4 dae6e705769dlde354270123cb11450e
fc60ac47683d7b8d0f811365565fd98c 4c8eb936bcab8d069fc33bd801b03ade
a2e1fbc5aa463d08ca19896d2bf59a07 1b851e6c239052172f296bfb5e724047
90a2181014f3b94a4e97d117b4381303 68cc39dbb2d198065ae3986547926cd2
162f40a29f0c3c8745c0f50fba3852e5 66d44575c29d39a03f0cda721984b6f4
40591f355e12d439ff150aab7613499d bd49adabc8676eef023b15b65bfc5ca0
6948109f23f350db82123535eb8a7433 bdabcb909271a6ecbcb58b936a88cd4e
8f2e6ff5800175f113253d8fa9ca8885 c2f552e657dc603f252e1a8e308f76f0
be79e2fb8f5d5fbbe2e30ecadd220723 c8c0aea8078cdfcb3868263ff8f09400
54da48781893a7e49ad5aff4af300cd8 04a6b6279ab3ff3afb64491c85194aab
760d58a606654f9f4400e8b38591356f bf6425aca26dc85244259ff2b19c41b9
f96f3ca9ecldde434da7d2d392b905dd f3dlf9af93dlaf5950bd493f5aa731b4
056df31bd267b6b90a079831aaf579be 0a39013137aac6d404f518cfd4684064
7e78bfe706ca4cf5e9c5453e9f7cfd2b 8b4c8d169a44e55c88d4a9a7f9474241
e221af44860018ab0856972e194cd934
```

A.3. Server Initial

The server sends the following payload in response, including an ACK frame, a CRYPTO frame, and no PADDING frames:

```
02000000000600405a020000560303ee fce7f7b37ba1d1632e96677825ddf739
88cfc79825df566dc5430b9a045a1200 130100002e00330024001d00209d3c94
0d89690b84d08a60993c144eca684d10 81287c834d5311bcf32bb9dala002b00
020304
```

The header from the server includes a new connection ID and a 2-byte packet number encoding for a packet number of 1:

```
c10000000010008f067a5502a4262b50040750001
```

As a result, after protection, the header protection sample is taken starting from the third protected byte:

```
sample = 2cd0991cd25b0aac406a5816b6394100
mask    = 2ec0d8356a
header  = cf0000000010008f067a5502a4262b5004075c0d9
```

The final protected packet is then:

```
cf0000000010008f067a5502a4262b500 4075c0d95a482cd0991cd25b0aac406a
5816b6394100f37a1c69797554780bb3 8cc5a99f5ede4cf73c3ec2493a1839b3
dbcba3f6ea46c5b7684df3548e7ddeb9 c3bf9c73cc3f3bded74b562bfb19fb84
022f8ef4cdd93795d77d06edbb7aaf2f 58891850abbdca3d20398c276456cbc4
2158407dd074ee
```

A.4. Retry

This shows a Retry packet that might be sent in response to the Initial packet in Appendix A.2. The integrity check includes the client-chosen connection ID value of 0x8394c8f03e515708, but that value is not included in the final Retry packet:

```
ff0000000010008f067a5502a4262b574 6f6b656e04a265ba2eff4d829058fb3f
0f2496ba
```

A.5. ChaCha20-Poly1305 Short Header Packet

This example shows some of the steps required to protect a packet with a short header. This example uses AEAD_CHACHA20_POLY1305.

In this example, TLS produces an application write secret from which a server uses HKDF-Expand-Label to produce four values: a key, an IV, a header protection key, and the secret that will be used after keys are updated (this last value is not used further in this example).

```
secret
  = 9ac312a7f877468ebe69422748ad00a1
  5443f18203a07d6060f688f30f21632b

key = HKDF-Expand-Label(secret, "quic key", "", 32)
  = c6d98ff3441c3fe1b2182094f69caa2e
  d4b716b65488960a7a984979fb23e1c8

iv  = HKDF-Expand-Label(secret, "quic iv", "", 12)
  = e0459b3474bdd0e44a41c144

hp  = HKDF-Expand-Label(secret, "quic hp", "", 32)
  = 25a282b9e82f06f21f488917a4fc8f1b
  73573685608597d0efcb076b0ab7a7a4

ku  = HKDF-Expand-Label(secret, "quic ku", "", 32)
  = 1223504755036d556342ee9361d25342
  1a826c9ecdf3c7148684b36b714881f9
```

The following shows the steps involved in protecting a minimal packet with an empty Destination Connection ID. This packet contains a single PING frame (that is, a payload of just 0x01) and has a packet number of 654360564. In this example, using a packet number of length 3 (that is, 49140 is encoded) avoids having to pad the payload of the packet; PADDING frames would be needed if the packet number is encoded on fewer bytes.

```
pn          = 654360564 (decimal)
nonce       = e0459b3474bdd0e46d417eb0
unprotected header = 4200bfff4
payload plaintext = 01
payload ciphertext = 655e5cd55c41f69080575d7999c25a5bfb
```

The resulting ciphertext is the minimum size possible. One byte is skipped to produce the sample for header protection.

```
sample = 5e5cd55c41f69080575d7999c25a5bfb
mask    = aefefe7d03
header  = 4cfe4189
```

The protected packet is the smallest possible packet size of 21 bytes.

```
packet = 4cfe4189655e5cd55c41f69080575d7999c25a5bfb
```

Appendix B. AEAD Algorithm Analysis

This section documents analyses used in deriving AEAD algorithm limits for AEAD_AES_128_GCM, AEAD_AES_128_CCM, and AEAD_AES_256_GCM. The analyses that follow use symbols for multiplication (*), division (/), and exponentiation (^), plus parentheses for establishing precedence. The following symbols are also used:

- t: The size of the authentication tag in bits. For these ciphers, t is 128.
- n: The size of the block function in bits. For these ciphers, n is 128.
- k: The size of the key in bits. This is 128 for AEAD_AES_128_GCM and AEAD_AES_128_CCM; 256 for AEAD_AES_256_GCM.
- l: The number of blocks in each packet (see below).
- q: The number of genuine packets created and protected by endpoints. This value is the bound on the number of packets that can be protected before updating keys.
- v: The number of forged packets that endpoints will accept. This value is the bound on the number of forged packets that an endpoint can reject before updating keys.
- o: The amount of offline ideal cipher queries made by an adversary.

The analyses that follow rely on a count of the number of block operations involved in producing each message. This analysis is performed for packets of size up to 2^{11} ($l = 2^7$) and 2^{16} ($l = 2^{12}$). A size of 2^{11} is expected to be a limit that matches common deployment patterns, whereas the 2^{16} is the maximum possible size of a QUIC packet. Only endpoints that strictly limit packet size can use the larger confidentiality and integrity limits that are derived using the smaller packet size.

For AEAD_AES_128_GCM and AEAD_AES_256_GCM, the message length (l) is the length of the associated data in blocks plus the length of the plaintext in blocks.

For AEAD_AES_128_CCM, the total number of block cipher operations is the sum of: the length of the associated data in blocks, the length of the ciphertext in blocks, the length of the plaintext in blocks, plus 1. In this analysis, this is simplified to a value of twice the length of the packet in blocks (that is, " $2l = 2^8$ " for packets that are limited to 2^{11} bytes, or " $2l = 2^{13}$ " otherwise). This

simplification is based on the packet containing all of the associated data and ciphertext. This results in a 1 to 3 block overestimation of the number of operations per packet.

B.1. Analysis of AEAD_AES_128_GCM and AEAD_AES_256_GCM Usage Limits

[GCM-MU] specify concrete bounds for AEAD_AES_128_GCM and AEAD_AES_256_GCM as used in TLS 1.3 and QUIC. This section documents this analysis using several simplifying assumptions:

- * The number of ciphertext blocks an attacker uses in forgery attempts is bounded by $v * l$, the number of forgery attempts and the size of each packet (in blocks).
- * The amount of offline work done by an attacker does not dominate other factors in the analysis.

The bounds in [GCM-MU] are tighter and more complete than those used in [AEBounds], which allows for larger limits than those described in [TLS13].

B.1.1. Confidentiality Limit

For confidentiality, Theorem (4.3) in [GCM-MU] establishes that - for a single user that does not repeat nonces - the dominant term in determining the distinguishing advantage between a real and random AEAD algorithm gained by an attacker is:

$$2 * (q * l)^2 / 2^n$$

For a target advantage of 2^{-57} , this results in the relation:

$$q \leq 2^{35} / l$$

Thus, endpoints that do not send packets larger than 2^{11} bytes cannot protect more than 2^{28} packets in a single connection without causing an attacker to gain an larger advantage than the target of 2^{-57} . The limit for endpoints that allow for the packet size to be as large as 2^{16} is instead 2^{23} .

B.1.2. Integrity Limit

For integrity, Theorem (4.3) in [GCM-MU] establishes that an attacker gains an advantage in successfully forging a packet of no more than:

$$(1 / 2^{(8 * n)}) + ((2 * v) / 2^{(2 * n)}) + ((2 * o * v) / 2^{(k + n)}) + (n * (v + (v * l)) / 2^k)$$

The goal is to limit this advantage to 2^{-57} . For AEAD_AES_128_GCM, the fourth term in this inequality dominates the rest, so the others can be removed without significant effect on the result. This produces the following approximation:

$$v \leq 2^{64} / l$$

Endpoints that do not attempt to remove protection from packets larger than 2^{11} bytes can attempt to remove protection from at most 2^{57} packets. Endpoints that do not restrict the size of processed packets can attempt to remove protection from at most 2^{52} packets.

For AEAD_AES_256_GCM, the same term dominates, but the larger value of k produces the following approximation:

$$v \leq 2^{192} / l$$

This is substantially larger than the limit for AEAD_AES_128_GCM. However, this document recommends that the same limit be applied to both functions as either limit is acceptably large.

B.2. Analysis of AEAD_AES_128_CCM Usage Limits

TLS [TLS13] and [AEBounds] do not specify limits on usage for AEAD_AES_128_CCM. However, any AEAD that is used with QUIC requires limits on use that ensure that both confidentiality and integrity are preserved. This section documents that analysis.

[CCM-ANALYSIS] is used as the basis of this analysis. The results of that analysis are used to derive usage limits that are based on those chosen in [TLS13].

For confidentiality, Theorem 2 in [CCM-ANALYSIS] establishes that an attacker gains a distinguishing advantage over an ideal pseudorandom permutation (PRP) of no more than:

$$(2l * q)^2 / 2^n$$

The integrity limit in Theorem 1 in [CCM-ANALYSIS] provides an attacker a strictly higher advantage for the same number of messages. As the targets for the confidentiality advantage and the integrity advantage are the same, only Theorem 1 needs to be considered.

Theorem 1 establishes that an attacker gains an advantage over an ideal PRP of no more than:

$$v / 2^t + (2l * (v + q))^2 / 2^n$$

As "t" and "n" are both 128, the first term is negligible relative to the second, so that term can be removed without a significant effect on the result.

This produces a relation that combines both encryption and decryption attempts with the same limit as that produced by the theorem for confidentiality alone. For a target advantage of 2^{-57} , this results in:

$$v + q \leq 2^{34.5} / 1$$

By setting "q = v", values for both confidentiality and integrity limits can be produced. Endpoints that limit packets to 2^{11} bytes therefore have both confidentiality and integrity limits of $2^{26.5}$ packets. Endpoints that do not restrict packet size have a limit of $2^{21.5}$.

Appendix C. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-tls-32

- * Added final values for Initial key derivation, Retry authentication, and TLS extension type for the QUIC Transport Parameters extension (#4431) (#4431)
- * Corrected rules for handling of 0-RTT (#4393, #4394)

C.2. Since draft-ietf-quic-tls-31

- * Packet protection limits are based on maximum-sized packets; improved analysis (#3701, #4175)

C.3. Since draft-ietf-quic-tls-30

- * Add a new error code for AEAD_LIMIT_REACHED code to avoid conflict (#4087, #4088)

C.4. Since draft-ietf-quic-tls-29

- * Updated limits on packet protection (#3788, #3789)
- * Allow for packet processing to continue while waiting for TLS to provide keys (#3821, #3874)

C.5. Since draft-ietf-quic-tls-28

- * Defined limits on the number of packets that can be protected with a single key and limits on the number of packets that can fail authentication (#3619, #3620)
- * Update Initial salt, Retry keys, and samples (#3711)

C.6. Since draft-ietf-quic-tls-27

- * Allowed CONNECTION_CLOSE in any packet number space, with restrictions on use of the application-specific variant (#3430, #3435, #3440)
- * Prohibit the use of the compatibility mode from TLS 1.3 (#3594, #3595)

C.7. Since draft-ietf-quic-tls-26

- * No changes

C.8. Since draft-ietf-quic-tls-25

- * No changes

C.9. Since draft-ietf-quic-tls-24

- * Rewrite key updates (#3050)
 - Allow but don't recommend deferring key updates (#2792, #3263)
 - More completely define received behavior (#2791)
 - Define the label used with HKDF-Expand-Label (#3054)

C.10. Since draft-ietf-quic-tls-23

- * Key update text update (#3050):
 - Recommend constant-time key replacement (#2792)
 - Provide explicit labels for key update key derivation (#3054)
- * Allow first Initial from a client to span multiple packets (#2928, #3045)
- * PING can be sent at any encryption level (#3034, #3035)

C.11. Since draft-ietf-quic-tls-22

- * Update the salt used for Initial secrets (#2887, #2980)

C.12. Since draft-ietf-quic-tls-21

- * No changes

C.13. Since draft-ietf-quic-tls-20

- * Mandate the use of the QUIC transport parameters extension (#2528, #2560)
- * Define handshake completion and confirmation; define clearer rules when it encryption keys should be discarded (#2214, #2267, #2673)

C.14. Since draft-ietf-quic-tls-18

- * Increased the set of permissible frames in 0-RTT (#2344, #2355)
- * Transport parameter extension is mandatory (#2528, #2560)

C.15. Since draft-ietf-quic-tls-17

- * Endpoints discard initial keys as soon as handshake keys are available (#1951, #2045)
- * Use of ALPN or equivalent is mandatory (#2263, #2284)

C.16. Since draft-ietf-quic-tls-14

- * Update the salt used for Initial secrets (#1970)
- * Clarify that TLS_AES_128_CCM_8_SHA256 isn't supported (#2019)
- * Change header protection
 - Sample from a fixed offset (#1575, #2030)
 - Cover part of the first byte, including the key phase (#1322, #2006)
- * TLS provides an AEAD and KDF function (#2046)
 - Clarify that the TLS KDF is used with TLS (#1997)
 - Change the labels for calculation of QUIC keys (#1845, #1971, #1991)

- * Initial keys are discarded once Handshake keys are available (#1951, #2045)
- C.17. Since draft-ietf-quic-tls-13
- * Updated to TLS 1.3 final (#1660)
- C.18. Since draft-ietf-quic-tls-12
- * Changes to integration of the TLS handshake (#829, #1018, #1094, #1165, #1190, #1233, #1242, #1252, #1450)
 - The cryptographic handshake uses CRYPTO frames, not stream 0
 - QUIC packet protection is used in place of TLS record protection
 - Separate QUIC packet number spaces are used for the handshake
 - Changed Retry to be independent of the cryptographic handshake
 - Limit the use of HelloRetryRequest to address TLS needs (like key shares)
 - * Changed codepoint of TLS extension (#1395, #1402)
- C.19. Since draft-ietf-quic-tls-11
- * Encrypted packet numbers.
- C.20. Since draft-ietf-quic-tls-10
- * No significant changes.
- C.21. Since draft-ietf-quic-tls-09
- * Cleaned up key schedule and updated the salt used for handshake packet protection (#1077)
- C.22. Since draft-ietf-quic-tls-08
- * Specify value for max_early_data_size to enable 0-RTT (#942)
 - * Update key derivation function (#1003, #1004)
- C.23. Since draft-ietf-quic-tls-07

- * Handshake errors can be reported with CONNECTION_CLOSE (#608, #891)
- C.24. Since draft-ietf-quic-tls-05
- No significant changes.
- C.25. Since draft-ietf-quic-tls-04
- * Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)
- C.26. Since draft-ietf-quic-tls-03
- No significant changes.
- C.27. Since draft-ietf-quic-tls-02
- * Updates to match changes in transport draft
- C.28. Since draft-ietf-quic-tls-01
- * Use TLS alerts to signal TLS errors (#272, #374)
 - * Require ClientHello to fit in a single packet (#338)
 - * The second client handshake flight is now sent in the clear (#262, #337)
 - * The QUIC header is included as AEAD Associated Data (#226, #243, #302)
 - * Add interface necessary for client address validation (#275)
 - * Define peer authentication (#140)
 - * Require at least TLS 1.3 (#138)
 - * Define transport parameters as a TLS extension (#122)
 - * Define handling for protected packets before the handshake completes (#39)
 - * Decouple QUIC version and ALPN (#12)
- C.29. Since draft-ietf-quic-tls-00
- * Changed bit used to signal key phase

- * Updated key phase markings during the handshake
- * Added TLS interface requirements section
- * Moved to use of TLS exporters for key derivation
- * Moved TLS error code definitions into this document

C.30. Since draft-thomson-quic-tls-01

- * Adopted as base for draft-ietf-quic-tls
- * Updated authors/editors list
- * Added status note

Contributors

The IETF QUIC Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document:

- * Adam Langley
- * Alessandro Ghedini
- * Christian Huitema
- * Christopher Wood
- * David Schinazi
- * Dragana Damjanovic
- * Eric Rescorla
- * Felix Guenther
- * Ian Swett
- * Jana Iyengar
- * (Kazuho Oku)
- * Marten Seemann
- * Martin Duke

- * Mike Bishop
- * Mikkel Fahnøe Jørgensen
- * Nick Banks
- * Nick Harper
- * Roberto Peon
- * Rui Paulo
- * Ryan Hamilton
- * Victor Vasiliev

Authors' Addresses

Martin Thomson (editor)
Mozilla

Email: mt@lowentropy.net

Sean Turner (editor)
sn3rd

Email: sean@sn3rd.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: September 14, 2017

J. Iyengar, Ed.
Google
M. Thomson, Ed.
Mozilla
March 13, 2017

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-02

Abstract

This document defines the core of the QUIC transport protocol. This document describes connection establishment, packet format, multiplexing and reliability. Accompanying documents describe the cryptographic handshake and loss detection.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic .

Working Group information can be found at <https://github.com/quicwg> ; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/transport> .

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
2. Conventions and Definitions	4
2.1. Notational Conventions	5
3. A QUIC Overview	5
3.1. Low-Latency Connection Establishment	6
3.2. Stream Multiplexing	6
3.3. Rich Signaling for Congestion Control and Loss Recovery	6
3.4. Stream and Connection Flow Control	6
3.5. Authenticated and Encrypted Header and Payload	7
3.6. Connection Migration and Resilience to NAT Rebinding	7
3.7. Version Negotiation	8
4. Versions	8
5. Packet Types and Formats	8
5.1. Long Header	9
5.2. Short Header	11
5.3. Version Negotiation Packet	12
5.4. Cleartext Packets	13
5.5. Encrypted Packets	14
5.6. Public Reset Packet	15
5.6.1. Public Reset Proof	15
5.7. Connection ID	16
5.8. Packet Numbers	16
5.8.1. Initial Packet Number	17
5.9. Handling Packets from Different Versions	17
6. Frames and Frame Types	18
7. Life of a Connection	19
7.1. Version Negotiation	19
7.1.1. Using Reserved Versions	20
7.2. Cryptographic and Transport Handshake	21
7.3. Transport Parameters	22
7.3.1. Transport Parameter Definitions	24

7.3.2.	Values of Transport Parameters for 0-RTT	24
7.3.3.	New Transport Parameters	25
7.3.4.	Version Negotiation Validation	25
7.4.	Proof of Source Address Ownership	27
7.4.1.	Client Address Validation Procedure	27
7.4.2.	Address Validation on Session Resumption	28
7.4.3.	Address Validation Token Integrity	29
7.5.	Connection Migration	29
7.6.	Connection Termination	30
8.	Frame Types and Formats	31
8.1.	STREAM Frame	31
8.2.	ACK Frame	32
8.2.1.	ACK Block Section	34
8.2.2.	Timestamp Section	35
8.2.3.	ACK Frames and Packet Protection	37
8.3.	WINDOW_UPDATE Frame	38
8.4.	BLOCKED Frame	39
8.5.	RST_STREAM Frame	39
8.6.	PADDING Frame	40
8.7.	PING frame	40
8.8.	CONNECTION_CLOSE frame	40
8.9.	GOAWAY Frame	41
9.	Packetization and Reliability	42
9.1.	Special Considerations for PMTU Discovery	44
10.	Streams: QUIC's Data Structuring Abstraction	45
10.1.	Life of a Stream	45
10.1.1.	idle	47
10.1.2.	open	47
10.1.3.	half-closed (local)	48
10.1.4.	half-closed (remote)	48
10.1.5.	closed	48
10.2.	Stream Identifiers	50
10.3.	Stream Concurrency	50
10.4.	Sending and Receiving Data	51
10.5.	Stream Prioritization	51
11.	Flow Control	52
11.1.	Edge Cases and Other Considerations	54
11.1.1.	Mid-stream RST_STREAM	54
11.1.2.	Response to a RST_STREAM	54
11.1.3.	Offset Increment	54
11.1.4.	BLOCKED frames	55
12.	Error Handling	55
12.1.	Connection Errors	55
12.2.	Stream Errors	56
12.3.	Error Codes	56
13.	Security and Privacy Considerations	60
13.1.	Spoofed ACK Attack	60
14.	IANA Considerations	61

14.1. QUIC Transport Parameter Registry	61
15. References	62
15.1. Normative References	62
15.2. Informative References	63
15.3. URIs	64
Appendix A. Contributors	64
Appendix B. Acknowledgments	64
Appendix C. Change Log	64
C.1. Since draft-ietf-quic-transport-01:	64
C.2. Since draft-ietf-quic-transport-00:	66
C.3. Since draft-hamilton-quic-transport-protocol-01:	67
Authors' Addresses	67

1. Introduction

QUIC is a multiplexed and secure transport protocol that runs on top of UDP. QUIC aims to provide a flexible set of features that allow it to be a general-purpose transport for multiple applications.

QUIC implements techniques learned from experience with TCP, SCTP and other transport protocols. Using UDP as the substrate, QUIC seeks to be compatible with legacy clients and middleboxes. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling. This allows the protocol to evolve without incurring a dependency on upgrades to middleboxes.

This document describes the core QUIC protocol, including the conceptual design, wire format, and mechanisms of the QUIC protocol for connection establishment, stream multiplexing, stream and connection-level flow control, and data reliability.

Accompanying documents describe QUIC's loss detection and congestion control [QUIC-RECOVERY], and the use of TLS 1.3 for key negotiation [QUIC-TLS].

2. Conventions and Definitions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when they are capitalized, they have the special meaning defined in [RFC2119].

Definitions of terms that are used in this document:

Client: The endpoint initiating a QUIC connection.

Server: The endpoint accepting incoming QUIC connections.

Endpoint: The client or server end of a connection.

Stream: A logical, bi-directional channel of ordered bytes within a QUIC connection.

Connection: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.

Connection ID: The identifier for a QUIC connection.

QUIC packet: A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

2.1. Notational Conventions

Packet and frame diagrams use the format described in [RFC2360] Section 3.1, with the following additional conventions:

[x] Indicates that x is optional

{x} Indicates that x is encrypted

x (A) Indicates that x is A bits long

x (A/B/C) ... Indicates that x is one of A, B, or C bits long

x (*) ... Indicates that x is variable-length

3. A QUIC Overview

This section briefly describes QUIC's key mechanisms and benefits. Key strengths of QUIC include:

- o Low-latency connection establishment
- o Multiplexing without head-of-line blocking
- o Authenticated and encrypted header and payload
- o Rich signaling for congestion control and loss recovery
- o Stream and connection flow control
- o Connection migration and resilience to NAT rebinding
- o Version negotiation

3.1. Low-Latency Connection Establishment

QUIC relies on a combined cryptographic and transport handshake for setting up a secure transport connection. QUIC connections are expected to commonly use 0-RTT handshakes, meaning that for most QUIC connections, data can be sent immediately following the client handshake packet, without waiting for a reply from the server. QUIC provides a dedicated stream (Stream ID 1) to be used for performing the cryptographic handshake and QUIC options negotiation. The format of the QUIC options and parameters used during negotiation are described in this document, but the handshake protocol that runs on Stream ID 1 is described in the accompanying cryptographic handshake draft [QUIC-TLS].

3.2. Stream Multiplexing

When application messages are transported over TCP, independent application messages can suffer from head-of-line blocking. When an application multiplexes many streams atop TCP's single-bytestream abstraction, a loss of a TCP segment results in blocking of all subsequent segments until a retransmission arrives, irrespective of the application streams that are encapsulated in subsequent segments. QUIC ensures that lost packets carrying data for an individual stream only impact that specific stream. Data received on other streams can continue to be reassembled and delivered to the application.

3.3. Rich Signaling for Congestion Control and Loss Recovery

QUIC's packet framing and acknowledgments carry rich information that help both congestion control and loss recovery in fundamental ways. Each QUIC packet carries a new packet number, including those carrying retransmitted data. This obviates the need for a separate mechanism to distinguish acknowledgments for retransmissions from those for original transmissions, avoiding TCP's retransmission ambiguity problem. QUIC acknowledgments also explicitly encode the delay between the receipt of a packet and its acknowledgment being sent, and together with the monotonically-increasing packet numbers, this allows for precise network roundtrip-time (RTT) calculation. QUIC's ACK frames support up to 256 ACK blocks, so QUIC is more resilient to reordering than TCP with SACK support, as well as able to keep more bytes on the wire when there is reordering or loss.

3.4. Stream and Connection Flow Control

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control mechanisms. At a high level, a QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data. As data is sent,

received, and delivered on a particular stream, the receiver sends WINDOW_UPDATE frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream. In addition to this stream-level flow control, QUIC implements connection-level flow control to limit the aggregate buffer that a QUIC receiver is willing to allocate to all streams on a connection. Connection-level flow control works in the same way as stream-level flow control, but the bytes delivered and highest received offset are all aggregates across all streams.

3.5. Authenticated and Encrypted Header and Payload

TCP headers appear in plaintext on the wire and are not authenticated, causing a plethora of injection and header manipulation issues for TCP, such as receive-window manipulation and sequence-number overwriting. While some of these are mechanisms used by middleboxes to improve TCP performance, others are active attacks. Even "performance-enhancing" middleboxes that routinely interpose on the transport state machine end up limiting the evolvability of the transport protocol, as has been observed in the design of MPTCP [RFC6824] and in its subsequent deployability issues.

Generally, QUIC packets are always authenticated and the payload is typically fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. Some early handshake packets, such as the Version Negotiation packet, are not encrypted, but information sent in these unencrypted handshake packets is later verified as part of cryptographic processing.

PUBLIC_RESET packets that reset a connection are currently not authenticated.

3.6. Connection Migration and Resilience to NAT Rebinding

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the client. QUIC's consistent connection ID allows connections to survive changes to the client's IP and port, such as those caused by NAT rebindings or by the client changing network connectivity to a new address. QUIC provides automatic cryptographic verification of a rebound client, since the client continues to use the same session key for encrypting and decrypting packets. The consistent connection ID can be used to allow migration of the connection to a new server IP address as well, since the Connection ID remains consistent across changes in the client's and the server's network addresses.

3.7. Version Negotiation

QUIC version negotiation allows for multiple versions of the protocol to be deployed and used concurrently. Version negotiation is described in Section 7.1.

4. Versions

QUIC versions are identified using a 32-bit value.

The version 0x00000000 is reserved to represent an invalid version. This version of the specification is identified by the number 0x00000001.

Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

Versions that follow the pattern 0x?a?a?a are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all octets is 1010 (in binary). A client or server MAY advertise support for any of these reserved versions.

Reserved version numbers will probably never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a server MAY advertise support for one of these versions and can expect that clients ignore the value.

[[RFC editor: please remove the remainder of this section before publication.]]

The version number for the final version of this specification (0x00000001), is reserved for the version of the protocol that is published as an RFC.

Version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, draft-ietf-quic-transport-13 would be identified as 0xff00000D.

Implementors are encouraged to register version numbers of QUIC that they are using for private experimentation on the github wiki [4].

5. Packet Types and Formats

We first describe QUIC's packet types and their formats, since some are referenced in subsequent mechanisms.

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. When discussing individual bits of fields, the least significant bit is referred to as bit 0. Hexadecimal notation is used for describing the value of fields.

Any QUIC packet has either a long or a short header, as indicated by the Header Form bit. Long headers are expected to be used early in the connection before version negotiation and establishment of 1-RTT keys, and for public resets. Short headers are minimal version-specific headers, which can be used after version negotiation and 1-RTT keys are established.

5.1. Long Header

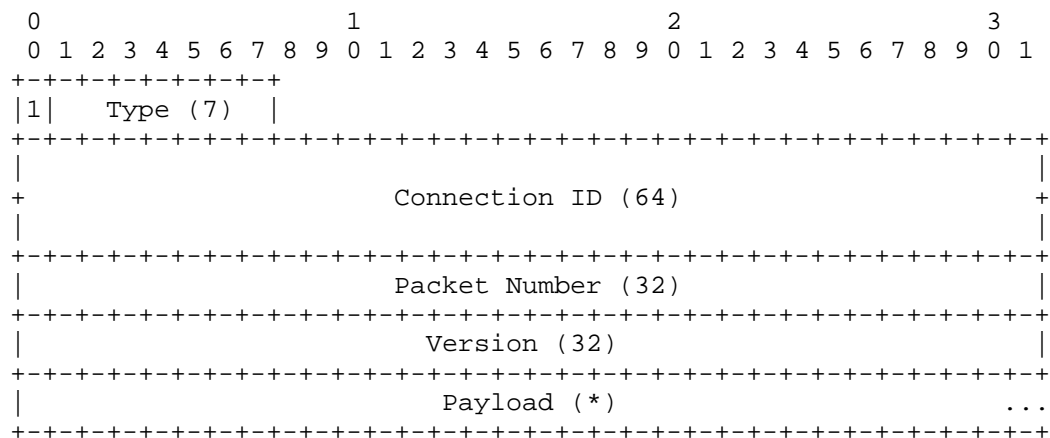


Figure 1: Long Header Format

Long headers are used for packets that are sent prior to the completion of version negotiation and establishment of 1-RTT keys. Once both conditions are met, a sender SHOULD switch to sending short-form headers. While inefficient, long headers MAY be used for packets encrypted with 1-RTT keys. The long form allows for special packets, such as the Version Negotiation and the Public Reset packets to be represented in this uniform fixed-length packet format. A long header contains the following fields:

Header Form: The most significant bit (0x80) of the first octet is set to 1 for long headers and 0 for short headers.

Long Packet Type: The remaining seven bits of first octet of a long packet is the packet type. This field can indicate one of 128 packet types. The types specified for this version are listed in Table 1.

Connection ID: Octets 1 through 8 contain the connection ID.
Section 5.7 describes the use of this field in more detail.

Packet Number: Octets 9 to 12 contain the packet number. {{packet-numbers}} describes the use of packet numbers.

Version: Octets 13 to 16 contain the selected protocol version.
This field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

Payload: Octets from 17 onwards (the rest of QUIC packet) are the payload of the packet.

The following packet types are defined:

Type	Name	Section
01	Version Negotiation	Section 5.3
02	Client Cleartext	Section 5.4
03	Non-Final Server Cleartext	Section 5.4
04	Final Server Cleartext	Section 5.4
05	0-RTT Encrypted	Section 5.5
06	1-RTT Encrypted (key phase 0)	Section 5.5
07	1-RTT Encrypted (key phase 1)	Section 5.5
08	Public Reset	Section 5.6

Table 1: Long Header Packet Types

The header form, packet type, connection ID, packet number and version fields of a long header packet are version-independent. The types of packets defined in Table 1 are version-specific. See Section 5.9 for details on how packets from different versions of QUIC are interpreted.

(TODO: Should the list of packet types be version-independent?)

The interpretation of the fields and the payload are specific to a version and packet type. Type-specific semantics for this version

are described in Section 5.3, Section 5.6, Section 5.4, and Section 5.5.

5.2. Short Header

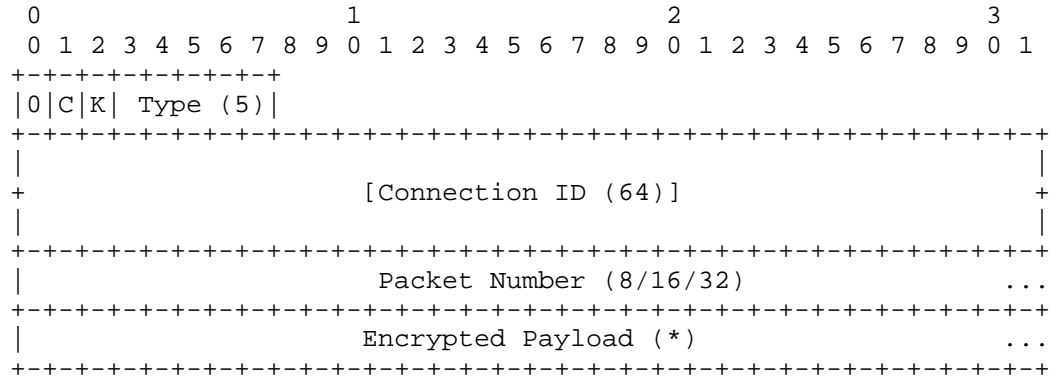


Figure 2: Short Header Format

The short header can be used after the version and 1-RTT keys are negotiated. This header form has the following fields:

Header Form: The most significant bit (0x80) of the first octet of a packet is the header form. This bit is set to 0 for the short header.

Connection ID Flag: The second bit (0x40) of the first octet indicates whether the Connection ID field is present. If set to 1, then the Connection ID field is present; if set to 0, the Connection ID field is omitted.

Key Phase Bit: The third bit (0x20) of the first octet indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details.

Short Packet Type: The remaining 5 bits of the first octet include one of 32 packet types. Table 2 lists the types that are defined for short packets.

Connection ID: If the Connection ID Flag is set, a connection ID occupies octets 1 through 8 of the packet. See Section 5.7 for more details.

Packet Number: The length of the packet number field depends on the packet type. This field can be 1, 2 or 4 octets long depending on the short packet type.

Encrypted Payload: Packets with a short header always include a 1-RTT protected payload.

The packet type in a short header currently determines only the size of the packet number field. Additional types can be used to signal the presence of other fields.

Type	Packet Number Size
01	1 octet
02	2 octets
03	4 octets

Table 2: Short Header Packet Types

The header form, connection ID flag and connection ID of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See Section 5.9 for details on how packets from different versions of QUIC are interpreted.

5.3. Version Negotiation Packet

A Version Negotiation packet is sent only by servers and is a response to a client packet of an unsupported version. It uses a long header and contains:

- o Octet 0: 0x81
- o Octets 1-8: Connection ID (echoed)
- o Octets 9-12: Packet Number (echoed)
- o Octets 13-16: Version (echoed)
- o Octets 17+: Payload

The payload of the Version Negotiation packet is a list of 32-bit versions which the server supports, as shown below.

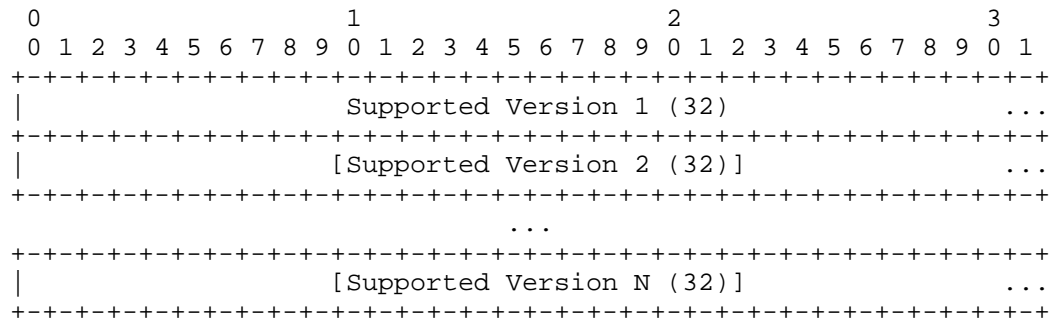


Figure 3: Version Negotiation Packet

See Section 7.1 for a description of the version negotiation process.

5.4. Cleartext Packets

Cleartext packets are sent during the handshake prior to key negotiation. A Client Cleartext packet contains:

- o Octet 0: 0x82
- o Octets 1-8: Connection ID (initial)
- o Octets 9-12: Packet number
- o Octets 13-16: Version
- o Octets 17+: Payload

Non-Final Server Cleartext packets contain:

- o Octet 0: 0x83
- o Octets 1-8: Connection ID (echoed)
- o Octets 9-12: Packet Number
- o Octets 13-16: Version
- o Octets 17+: Payload

Final Server Cleartext packets contains:

- o Octet 0: 0x84
- o Octets 1-8: Connection ID (final)

- o Octets 9-12: Packet Number
- o Octets 13-16: Version
- o Octets 17+: Payload

The client MUST choose a random 64-bit value and use it as the initial Connection ID in all packets until the server replies with the final Connection ID. The server echoes the client's Connection ID in Non-Final Server Cleartext packets. The first Final Server Cleartext and all subsequent packets MUST use the final Connection ID, as described in Section 5.7.

The payload of a Cleartext packet consists of a sequence of frames, as described in Section 6.

(TODO: Add hash before frames.)

5.5. Encrypted Packets

Packets encrypted with either 0-RTT or 1-RTT keys may be sent with long headers. Different packet types explicitly indicate the encryption level for ease of decryption. These packets contain:

- o Octet 0: 0x85, 0x86 or 0x87
- o Octets 1-8: Connection ID (initial or final)
- o Octets 9-12: Packet Number
- o Octets 13-16: Version
- o Octets 17+: Encrypted Payload

A first octet of 0x85 indicates a 0-RTT packet. After the 1-RTT keys are established, key phases are used by the QUIC packet protection to identify the correct packet protection keys. The initial key phase is 0. See [QUIC-TLS] for more details.

The encrypted payload is both authenticated and encrypted using packet protection keys. [QUIC-TLS] describes packet protection in detail. After decryption, the plaintext consists of a sequence of frames, as described in Section 6.

5.6. Public Reset Packet

A Public Reset packet is only sent by servers and is used to abruptly terminate communications. Public Reset is provided as an option of last resort for a server that does not have access to the state of a connection. This is intended for use by a server that has lost state (for example, through a crash or outage). A server that wishes to communicate a fatal connection error **MUST** use a `CONNECTION_CLOSE` frame if it has sufficient state to do so.

A Public Reset packet contains:

- o Octet 0: 0x88
- o Octets 1-8: Echoed data (octets 1-8 of received packet)
- o Octets 9-12: Echoed data (octets 9-12 of received packet)
- o Octets 13-16: Version
- o Octets 17+: Public Reset Proof

For a client that sends a connection ID on every packet, the Connection ID field is simply an echo of the initial Connection ID, and the Packet Number field includes an echo of the client's packet number (and, depending on the client's packet number length, 0, 2, or 3 additional octets from the client's packet).

A Public Reset packet sent by a server indicates that it does not have the state necessary to continue with a connection. In this case, the server will include the fields that prove that it originally participated in the connection (see Section 5.6.1 for details).

Upon receipt of a Public Reset packet that contains a valid proof, a client **MUST** tear down state associated with the connection. The client **MUST** then cease sending packets on the connection and **SHOULD** discard any subsequent packets that arrive. A Public Reset that does not contain a valid proof **MUST** be ignored.

5.6.1. Public Reset Proof

TODO: Details to be added.

5.7. Connection ID

QUIC connections are identified by their 64-bit Connection ID. All long headers contain a Connection ID. Short headers indicate the presence of a Connection ID using the CONNECTION_ID flag. When present, the Connection ID is in the same location in all packet headers, making it straightforward for middleboxes, such as load balancers, to locate and use it.

When a connection is initiated, the client MUST choose a random value and use it as the initial Connection ID until the final value is available. The initial Connection ID is a suggestion to the server. The server echoes this value in all packets until the handshake is successful (see [QUIC-TLS]). On a successful handshake, the server MUST select the final Connection ID for the connection and use it in Final Server Cleartext packets. This final Connection ID MAY be the one proposed by the client or MAY be a new server-selected value. All subsequent packets from the server MUST contain this value. On handshake completion, the client MUST switch to using the final Connection ID for all subsequent packets.

Thus, all Client Cleartext packets, 0-RTT Encrypted packets, and Non-Final Server Cleartext packets MUST use the client's randomly-generated initial Connection ID. Final Server Cleartext packets, 1-RTT Encrypted packets, and all short-header packets MUST use the final Connection ID.

5.8. Packet Numbers

The packet number is a 64-bit unsigned number and is used as part of a cryptographic nonce for packet encryption. Each endpoint maintains a separate packet number for sending and receiving. The packet number for sending MUST increase by at least one after sending any packet.

A QUIC endpoint MUST NOT reuse a packet number within the same connection (that is, under the same cryptographic keys). If the packet number for sending reaches $2^{64} - 1$, the sender MUST close the connection by sending a CONNECTION_CLOSE frame with the error code QUIC_SEQUENCE_NUMBER_LIMIT_REACHED (connection termination is described in Section 7.6.)

To reduce the number of bits required to represent the packet number over the wire, only the least significant bits of the packet number are transmitted over the wire, up to 32 bits. The actual packet number for each packet is reconstructed at the receiver based on the largest packet number received on a successfully authenticated packet.

A packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. For example, if the highest successfully authenticated packet had a packet number of 0xaa82f30e, then a packet containing a 16-bit value of 0x1f94 will be decoded as 0xaa831f94.

The sender **MUST** use a packet number size able to represent more than twice as large a range than the difference between the largest acknowledged packet and packet number being sent. A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received. An endpoint **MAY** use a larger packet number size to safeguard against such reordering.

As a result, the size of the packet number encoding is at least one more than the base 2 logarithm of the number of contiguous unacknowledged packet numbers, including the new packet.

For example, if an endpoint has received an acknowledgment for packet 0x6afa2f, sending a packet with a number of 0x6b4264 requires a 16-bit or larger packet number encoding; whereas a 32-bit packet number is needed to send a packet with a number of 0x6bc107.

5.8.1. Initial Packet Number

The initial value for packet number **MUST** be a 31-bit random number. That is, the value is selected from an uniform random distribution between 0 and $2^{31}-1$. [RFC4086] provides guidance on the generation of random values.

The first set of packets sent by an endpoint **MUST** include the low 32-bits of the packet number. Once any packet has been acknowledged, subsequent packets can use a shorter packet number encoding.

5.9. Handling Packets from Different Versions

Between different versions the following things are guaranteed to remain constant:

- o the location of the header form flag,
- o the location of the Connection ID flag in short headers,
- o the location and size of the Connection ID field in both header forms,
- o the location and size of the Version field in long headers, and

- o the location and size of the Packet Number field in long headers.

Implementations **MUST** assume that an unsupported version uses an unknown packet format. All other fields **MUST** be ignored when processing a packet that contains an unsupported version.

6. Frames and Frame Types

The payload of cleartext packets and the plaintext after decryption of encrypted payloads consists of a sequence of frames, as shown in Figure 4.

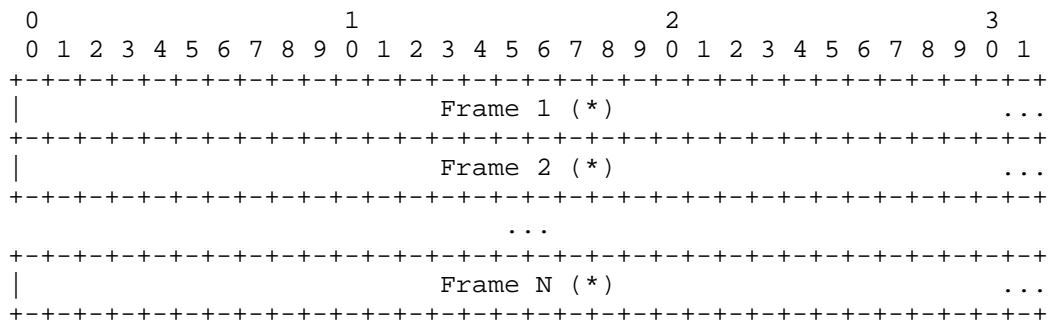


Figure 4: Contents of Encrypted Payload

Encrypted payloads **MUST** contain at least one frame, and **MAY** contain multiple frames and multiple frame types.

Frames **MUST** fit within a single QUIC packet and **MUST NOT** span a QUIC packet boundary. Each frame begins with a Frame Type byte, indicating its type, followed by additional type-dependent fields:

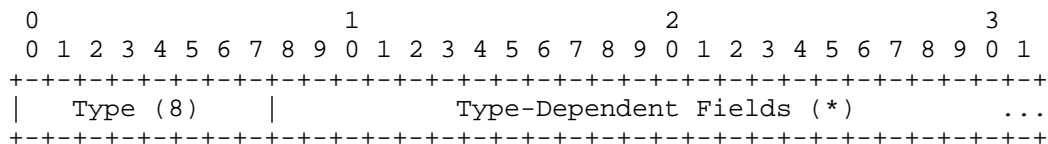


Figure 5: Generic Frame Layout

Frame types are listed in Table 3. Note that the Frame Type byte in STREAM and ACK frames is used to carry other frame-specific flags. For all other frames, the Frame Type byte simply identifies the frame. These frames are explained in more detail as they are referenced later in the document.

Type-field value	Frame type	Definition
0x00	PADDING	Section 8.6
0x01	RST_STREAM	Section 8.5
0x02	CONNECTION_CLOSE	Section 8.8
0x03	GOAWAY	Section 8.9
0x04	WINDOW_UPDATE	Section 8.3
0x05	BLOCKED	Section 8.4
0x07	PING	Section 8.7
0x40 - 0x7f	ACK	Section 8.2
0x80 - 0xff	STREAM	Section 8.1

Table 3: Frame Types

7. Life of a Connection

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment intertwines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency, as described in Section 7.2. Once established, a connection may migrate to a different IP or port at either endpoint, due to NAT rebinding or mobility, as described in Section 7.5. Finally a connection may be terminated by either endpoint, as described in Section 7.6.

7.1. Version Negotiation

QUIC's connection establishment begins with version negotiation, since all communication between the endpoints, including packet and frame formats, relies on the two endpoints agreeing on a version.

A QUIC connection begins with a client sending a handshake packet. The details of the handshake mechanisms are described in Section 7.2, but all of the initial packets sent from the client to the server MUST use the long header format and MUST specify the version of the protocol being used.

When the server receives a packet from a client with the long header format, it compares the client's version to the versions it supports.

If the version selected by the client is not acceptable to the server, the server discards the incoming packet and responds with a Version Negotiation packet (Section 5.3). This includes a list of versions that the server will accept. A server **MUST** send a Version Negotiation packet for every packet that it receives with an unacceptable version.

If the packet contains a version that is acceptable to the server, the server proceeds with the handshake (Section 7.2). This commits the server to the version that the client selected.

When the client receives a Version Negotiation packet from the server, it should select an acceptable protocol version. If the server lists an acceptable version, the client selects that version and reattempts to create a connection using that version. Though the contents of a packet might not change in response to version negotiation, a client **MUST** increase the packet number it uses on every packet it sends. Packets **MUST** continue to use long headers and **MUST** include the new negotiated protocol version.

The client **MUST** use the long header format and include its selected version on all packets until it has 1-RTT keys and it has received a packet from the server which is not a Version Negotiation packet.

A client **MUST NOT** change the version it uses unless it is in response to a Version Negotiation packet from the server. Once a client receives a packet from the server which is not a Version Negotiation packet, it **MUST** ignore Version Negotiation packets on the same connection.

Version negotiation uses unprotected data. The result of the negotiation **MUST** be revalidated as part of the cryptographic handshake (see Section 7.3.4).

7.1.1. Using Reserved Versions

For a server to use a new version in the future, clients must correctly handle unsupported versions. To help ensure this, a server **SHOULD** include a reserved version (see Section 4) while generating a Version Negotiation packet.

The design of version negotiation permits a server to avoid maintaining state for packets that it rejects in this fashion. However, when the server generates a Version Negotiation packet, it cannot randomly generate a reserved version number. This is because

the server is required to include the same value in its transport parameters (see Section 7.3.4). To avoid the selected version number changing during connection establishment, the reserved version SHOULD be generated as a function of values that will be available to the server when later generating its handshake packets.

A pseudorandom function that takes client address information (IP and port) and the client selected version as input would ensure that there is sufficient variability in the values that a server uses.

A client MAY send a packet using a reserved version number. This can be used to solicit a list of supported versions from a server.

7.2. Cryptographic and Transport Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC allocates stream 1 for the cryptographic handshake. This version of QUIC uses TLS 1.3 [QUIC-TLS].

QUIC provides this stream with reliable, ordered delivery of data. In return, the cryptographic handshake provides QUIC with:

- o authenticated key exchange, where
 - * a server is always authenticated,
 - * a client is optionally authenticated,
 - * every connection produces distinct and unrelated keys,
 - * keying material is usable for packet protection for both 0-RTT and 1-RTT packets, and
 - * 1-RTT keys have forward secrecy
- o authenticated values for the transport parameters of the peer (see Section 7.3)
- o authenticated confirmation of version negotiation (see Section 7.3.4)
- o authenticated negotiation of an application protocol (TLS uses ALPN [RFC7301] for this purpose)
- o for the server, the ability to carry data that provides assurance that the client can receive packets that are addressed with the transport address that is claimed by the client (see Section 7.4)

The initial cryptographic handshake message MUST be sent in a single packet. Any second attempt that is triggered by address validation MUST also be sent within a single packet. This avoids having to reassemble a message from multiple packets. Reassembling messages requires that a server maintain state prior to establishing a connection, exposing the server to a denial of service risk.

The first client packet of the cryptographic handshake protocol MUST fit within a 1280 octet QUIC packet. This includes overheads that reduce the space available to the cryptographic handshake protocol.

Details of how TLS is integrated with QUIC is provided in more detail in [QUIC-TLS].

7.3. Transport Parameters

During connection establishment, both endpoints make authenticated declarations of their transport parameters. These declarations are made unilaterally by each endpoint. Endpoints are required to comply with the restrictions implied by these parameters; the description of each parameter includes rules for its handling.

The format of the transport parameters is the TransportParameters struct from Figure 6. This is described using the presentation language from Section 3 of [I-D.ietf-tls-tls13].

```
uint32 QuicVersion;

enum {
    stream_fc_offset(0),
    connection_fc_offset(1),
    concurrent_streams(2),
    idle_timeout(3),
    truncate_connection_id(4),
    (65535)
} TransportParameterId;

struct {
    TransportParameterId parameter;
    opaque value<0..2^16-1>;
} TransportParameter;

struct {
    select (Handshake.msg_type) {
        case client_hello:
            QuicVersion negotiated_version;
            QuicVersion initial_version;

            case encrypted_extensions:
                QuicVersion supported_versions<2..2^8-4>;
    };
    TransportParameter parameters<30..2^16-1>;
} TransportParameters;
```

Figure 6: Definition of TransportParameters

The "extension_data" field of the quic_transport_parameters extension defined in [QUIC-TLS] contains a TransportParameters value. TLS encoding rules are therefore used to encode the transport parameters.

QUIC encodes transport parameters into a sequence of octets, which are then included in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the value provided by its peer. In particular, version negotiation MUST be validated (see Section 7.3.4) before the connection establishment is considered properly complete.

Definitions for each of the defined transport parameters are included in Section 7.3.1.

7.3.1. Transport Parameter Definitions

An endpoint **MUST** include the following parameters in its encoded `TransportParameters`:

`stream_fc_offset` (0x0000): The initial stream level flow control offset parameter is encoded as an unsigned 32-bit integer in units of octets. The sender of this parameter indicates that the flow control offset for all stream data sent toward it is this value.

`connection_fc_offset` (0x0001): The connection level flow control offset parameter contains the initial connection flow control window encoded as an unsigned 32-bit integer in units of 1024 octets. That is, the value here is multiplied by 1024 to determine the actual flow control offset. The sender of this parameter sets the byte offset for connection level flow control to this value. This is equivalent to sending a `WINDOW_UPDATE` (Section 8.3) for the connection immediately after completing the handshake.

`concurrent_streams` (0x0002): The maximum number of concurrent streams parameter is encoded as an unsigned 32-bit integer.

`idle_timeout` (0x0003): The idle timeout is a value in seconds that is encoded as an unsigned 16-bit integer. The maximum value is 600 seconds (10 minutes).

An endpoint **MAY** use the following transport parameters:

`truncate_connection_id` (0x0004): The truncated connection identifier parameter indicates that packets sent to the peer can omit the connection ID. This can be used by an endpoint where the 5-tuple is sufficient to identify a connection. This parameter is zero length. Omitting the parameter indicates that the endpoint relies on the connection ID being present in every packet.

7.3.2. Values of Transport Parameters for 0-RTT

Transport parameters from the server **SHOULD** be remembered by the client for use with 0-RTT data. A client that doesn't remember values from a previous connection can instead assume the following values: `stream_fc_offset` (65535), `connection_fc_offset` (65535), `concurrent_streams` (10), `idle_timeout` (600), `truncate_connection_id` (absent).

If assumed values change as a result of completing the handshake, the client is expected to respect the new values. This introduces some

potential problems, particularly with respect to transport parameters that establish limits:

- o A client might exceed a newly declared connection or stream flow control limit with 0-RTT data. If this occurs, the client ceases transmission as though the flow control limit was reached. Once WINDOW_UPDATE frames indicating an increase to the affected flow control offsets is received, the client can recommence sending.
- o Similarly, a client might exceed the concurrent stream limit declared by the server. A client MUST reset any streams that exceed this limit. A server SHOULD reset any streams it cannot handle with a code that allows the client to retry any application action bound to those streams.

A server MAY close a connection if remembered or assumed 0-RTT transport parameters cannot be supported, using an error code that is appropriate to the specific condition. For example, a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA might be used to indicate that exceeding flow control limits caused the error. A client that has a connection closed due to an error condition SHOULD NOT attempt 0-RTT when attempting to create a new connection.

7.3.3. New Transport Parameters

New transport parameters can be used to negotiate new protocol behavior. An endpoint MUST ignore transport parameters that it does not support. Absence of a transport parameter therefore disables any optional protocol feature that is negotiated using the parameter.

The definition of a transport parameter SHOULD include a default value that a client can use when establishing a new connection. If no default is specified, the value can be assumed to be absent when attempting 0-RTT.

New transport parameters can be registered according to the rules in Section 14.1.

7.3.4. Version Negotiation Validation

The transport parameters include three fields that encode version information. These retroactively authenticate the version negotiation (see Section 7.1) that is performed prior to the cryptographic handshake.

The cryptographic handshake provides integrity protection for the negotiated version as part of the transport parameters (see

Section 7.3). As a result, modification of version negotiation packets by an attacker can be detected.

The client includes two fields in the transport parameters:

- o The `negotiated_version` is the version that was finally selected for use. This MUST be identical to the value that is on the packet that carries the ClientHello. A server that receives a `negotiated_version` that does not match the version of QUIC that is in use MUST terminate the connection with a `QUIC_VERSION_NEGOTIATION_MISMATCH` error code.
- o The `initial_version` is the version that the client initially attempted to use. If the server did not send a version negotiation packet Section 5.3, this will be identical to the `negotiated_version`.

A server that processes all packets in a stateful fashion can remember how version negotiation was performed and validate the `initial_version` value.

A server that does not maintain state for every packet it receives (i.e., a stateless server) uses a different process. If the initial and negotiated versions are the same, a stateless server can accept the value.

If the initial version is different from the `negotiated_version`, a stateless server MUST check that it would have sent a version negotiation packet if it had received a packet with the indicated `initial_version`. If a server would have accepted the version included in the `initial_version` and the value differs from the value of `negotiated_version`, the server MUST terminate the connection with a `QUIC_VERSION_NEGOTIATION_MISMATCH` error.

The server includes a list of versions that it would send in any version negotiation packet (Section 5.3) in `supported_versions`. This value is set even if it did not send a version negotiation packet.

The client can validate that the `negotiated_version` is included in the `supported_versions` list and - if version negotiation was performed - that it would have selected the negotiated version. A client MUST terminate the connection with a `QUIC_VERSION_NEGOTIATION_MISMATCH` error code if the `negotiated_version` value is not included in the `supported_versions` list. A client MUST terminate with a `QUIC_VERSION_NEGOTIATION_MISMATCH` error code if version negotiation occurred but it would have selected a different version based on the value of the `supported_versions` list.

7.4. Proof of Source Address Ownership

Transport protocols commonly spend a round trip checking that a client owns the transport address (IP and port) that it claims. Verifying that a client can receive packets sent to its claimed transport address protects against spoofing of this information by malicious clients.

This technique is used primarily to avoid QUIC from being used for traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

Several methods are used in QUIC to mitigate this attack. Firstly, the initial handshake packet from a client is padded to at least 1280 octets. This allows a server to send a similar amount of data without risking causing an amplification attack toward an unproven remote address.

A server eventually confirms that a client has received its messages when the cryptographic handshake successfully completes. This might be insufficient, either because the server wishes to avoid the computational cost of completing the handshake, or it might be that the size of the packets that are sent during the handshake is too large. This is especially important for 0-RTT, where the server might wish to provide application data traffic - such as a response to a request - in response to the data carried in the early data from the client.

To send additional data prior to completing the cryptographic handshake, the server then needs to validate that the client owns the address that it claims.

Source address validation is therefore performed during the establishment of a connection. TLS provides the tools that support the feature, but basic validation is performed by the core transport protocol.

7.4.1. Client Address Validation Procedure

QUIC uses token-based address validation. Any time the server wishes to validate a client address, it provides the client with a token. As long as the token cannot be easily guessed (see Section 7.4.3), if the client is able to return that token, it proves to the server that it received the token.

During the processing of the cryptographic handshake messages from a client, TLS will request that QUIC make a decision about whether to proceed based on the information it has. TLS will provide QUIC with any token that was provided by the client. For an initial packet, QUIC can decide to abort the connection, allow it to proceed, or request address validation.

If QUIC decides to request address validation, it provides the cryptographic handshake with a token. The contents of this token are consumed by the server that generates the token, so there is no need for a single well-defined format. A token could include information about the claimed client address (IP and port), a timestamp, and any other supplementary information the server will need to validate the token in the future.

The cryptographic handshake is responsible for enacting validation by sending the address validation token to the client. A legitimate client will include a copy of the token when it attempts to continue the handshake. The cryptographic handshake extracts the token then asks QUIC a second time whether the token is acceptable. In response, QUIC can either abort the connection or permit it to proceed.

A connection MAY be accepted without address validation - or with only limited validation - but a server SHOULD limit the data it sends toward an unvalidated address. Successful completion of the cryptographic handshake implicitly provides proof that the client has received packets from the server.

7.4.2. Address Validation on Session Resumption

A server MAY provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

A different type of token is needed when resuming. Unlike the token that is created during a handshake, there might be some time between when the token is created and when the token is subsequently used. Thus, a resumption token SHOULD include an expiration time. It is also unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

This token can be provided to the cryptographic handshake immediately after establishing a connection. QUIC might also generate an updated token if significant time passes or the client address changes for

any reason (see Section 7.5). The cryptographic handshake is responsible for providing the client with the token. In TLS the token is included in the ticket that is used for resumption and 0-RTT, which is carried in a NewSessionTicket message.

7.4.3. Address Validation Token Integrity

An address validation token **MUST** be difficult to guess. Including a large enough random value in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token **MUST** be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

In TLS the address validation token is often bundled with the information that TLS requires, such as the resumption secret. In this case, adding integrity protection can be delegated to the cryptographic handshake protocol, avoiding redundant protection. If integrity protection is delegated to the cryptographic handshake, an integrity failure will result in immediate cryptographic handshake failure. If integrity protection is performed by QUIC, QUIC **MUST** abort the connection if the integrity check fails with a `QUIC_ADDRESS_VALIDATION_FAILURE` error code.

7.5. Connection Migration

QUIC connections are identified by their 64-bit Connection ID. QUIC's consistent connection ID allows connections to survive changes to the client's IP and/or port, such as those caused by client or server migrating to a new network. QUIC also provides automatic cryptographic verification of a client which has changed its IP address because the client continues to use the same session key for encrypting and decrypting packets.

DISCUSS: Simultaneous migration. Is this reasonable?

TODO: Perhaps move mitigation techniques from Security Considerations here.

7.6. Connection Termination

Connections should remain open until they become idle for a pre-negotiated period of time. A QUIC connection, once established, can be terminated in one of three ways:

1. **Explicit Shutdown:** An endpoint sends a `CONNECTION_CLOSE` frame to initiate a connection termination. An endpoint may send a `GOAWAY` frame to the peer prior to a `CONNECTION_CLOSE` to indicate that the connection will soon be terminated. A `GOAWAY` frame signals to the peer that any active streams will continue to be processed, but the sender of the `GOAWAY` will not initiate any additional streams and will not accept any new incoming streams. On termination of the active streams, a `CONNECTION_CLOSE` may be sent. If an endpoint sends a `CONNECTION_CLOSE` frame while unterminated streams are active (no `FIN` bit or `RST_STREAM` frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.
2. **Implicit Shutdown:** The default idle timeout for a QUIC connection is 30 seconds, and is a required parameter in connection negotiation. The maximum is 10 minutes. If there is no network activity for the duration of the idle timeout, the connection is closed. By default a `CONNECTION_CLOSE` frame will be sent. A silent close option can be enabled when it is expensive to send an explicit close, such as mobile networks that must wake up the radio.
3. **Abrupt Shutdown:** An endpoint may send a Public Reset packet at any time during the connection to abruptly terminate an active connection. A Public Reset packet **SHOULD** only be used as a final recourse. Commonly, a public reset is expected to be sent when a packet on an established connection is received by an endpoint that is unable to decrypt the packet. For instance, if a server reboots mid-connection and loses any cryptographic state associated with open connections, and then receives a packet on an open connection, it should send a Public Reset packet in return. (TODO: articulate rules around when a public reset should be sent.)

TODO: Connections that are terminated are added to a `TIME_WAIT` list at the server, so as to absorb any straggler packets in the network. Discuss `TIME_WAIT` list.

8. Frame Types and Formats

As described in Section 6, Regular packets contain one or more frames. We now describe the various QUIC frame types that can be present in a Regular packet. The use of these frames and various frame header bits are described in subsequent sections.

8.1. STREAM Frame

STREAM frames implicitly create a stream and carry stream data. The type byte for a STREAM frame contains embedded flags, and is formatted as "1FDOOOSS". These bits are parsed as follows:

- o The leftmost bit must be set to 1, indicating that this is a STREAM frame.
- o "F" is the FIN bit, which is used for stream termination.
- o The "D" bit indicates whether a Data Length field is present in the STREAM header. When set to 0, this field indicates that the Stream Data field extends to the end of the packet. When set to 1, this field indicates that Data Length field contains the length (in bytes) of the Stream Data field. The option to omit the length should only be used when the packet is a "full-sized" packet, to avoid the risk of corruption via padding.
- o The "OOO" bits encode the length of the Offset header field as 0, 16, 24, 32, 40, 48, 56, or 64 bits long.
- o The "SS" bits encode the length of the Stream ID header field as 8, 16, 24, or 32 bits. (DISCUSS: Consider making this 8, 16, 32, 64.)

A STREAM frame is shown below.

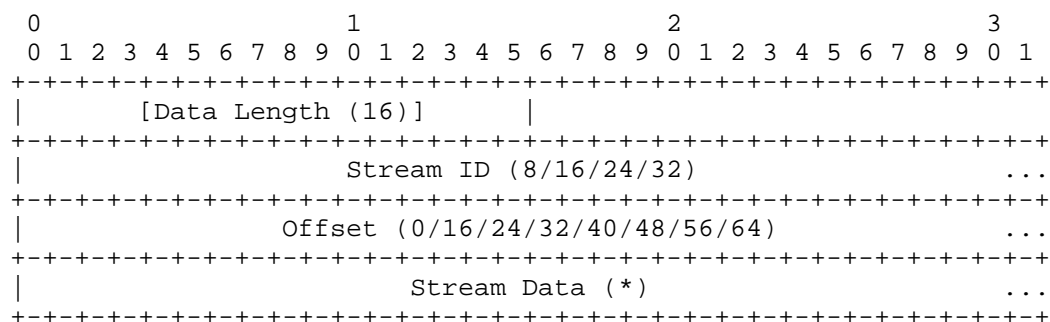


Figure 7: STREAM Frame Format

The STREAM frame contains the following fields:

Data Length: An optional 16-bit unsigned number specifying the length of the Stream Data field in this STREAM frame. This field is present when the "D" bit is set to 1.

Stream ID: A variable-sized unsigned ID unique to this stream.

Offset: A variable-sized unsigned number specifying the byte offset in the stream for the data in this STREAM frame. The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the re-constructed offset and data length - MUST be less than 2^{64} .

Stream Data: The bytes from the designated stream to be delivered.

A STREAM frame MUST have either non-zero data length or the FIN bit set.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet MAY bundle STREAM frames from multiple streams.

Implementation note: One of the benefits of QUIC is avoidance of head-of-line blocking across multiple streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Note that when data from multiple streams is bundled into a single QUIC packet, loss of that packet blocks all those streams from making progress. An implementation is therefore advised to bundle as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

8.2. ACK Frame

Receivers send ACK frames to inform senders which packets they have received and processed, as well as which packets are considered missing. The ACK frame contains between 1 and 256 ACK blocks. ACK blocks are ranges of acknowledged packets.

To limit ACK blocks to those that have not yet been received by the sender, the receiver SHOULD track which ACK frames have been acknowledged by its peer. Once an ACK frame has been acknowledged, the packets it acknowledges SHOULD not be acknowledged again. To handle cases where the receiver is only sending ACK frames, and hence will not receive acknowledgments for its packets, it MAY send a PING frame at most once per RTT to explicitly request acknowledgment.

To limit receiver state or the size of ACK frames, a receiver MAY limit the number of ACK blocks it sends. A receiver can do this even without receiving acknowledgment of its ACK frames, with the knowledge this could cause the sender to unnecessarily retransmit some data.

Unlike TCP SACKs, QUIC ACK blocks are cumulative and therefore irrevocable. Once a packet has been acknowledged, even if it does not appear in a future ACK frame, it is assumed to be acknowledged.

QUIC ACK frames contain a timestamp section with up to 255 timestamps. Timestamps enable better congestion control, but are not required for correct loss recovery, and old timestamps are less valuable, so it is not guaranteed every timestamp will be received by the sender. A receiver SHOULD send a timestamp exactly once for each received packet containing retransmittable frames. A receiver MAY send timestamps for non-retransmittable packets.

A sender MAY intentionally skip packet numbers to introduce entropy into the connection, to avoid opportunistic acknowledgement attacks. The sender MUST close the connection if an unsent packet number is acknowledged. The format of the ACK frame is efficient at expressing blocks of missing packets; skipping packet numbers between 1 and 255 effectively provides up to 8 bits of efficient entropy on demand, which should be adequate protection against most opportunistic acknowledgement attacks.

The type byte for a ACK frame contains embedded flags, and is formatted as "01NULLMM". These bits are parsed as follows:

- o The first two bits must be set to 01 indicating that this is an ACK frame.
- o The "N" bit indicates whether the frame has more than 1 range of acknowledged packets (i.e., whether the ACK Block Section contains a Num Blocks field).
- o The "U" bit is unused and MUST be set to zero.
- o The two "LL" bits encode the length of the Largest Acknowledged field as 1, 2, 4, or 6 bytes long.
- o The two "MM" bits encode the length of the ACK Block Length fields as 1, 2, 4, or 6 bytes long.

An ACK frame is shown below.

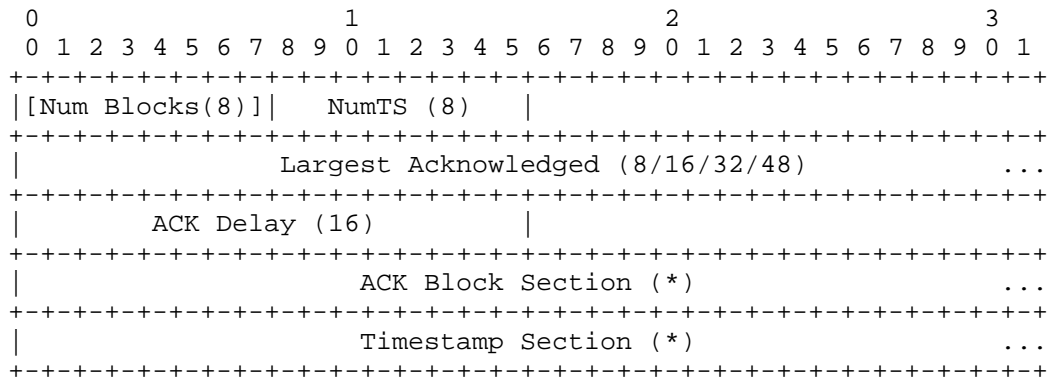


Figure 8: ACK Frame Format

The fields in the ACK frame are as follows:

Num Blocks (opt): An optional 8-bit unsigned value specifying the number of additional ACK blocks (besides the required First ACK Block) in this ACK frame. Only present if the 'N' flag bit is 1.

Num Timestamps: An unsigned 8-bit number specifying the total number of <packet number, timestamp> pairs in the Timestamp Section.

Largest Acknowledged: A variable-sized unsigned value representing the largest packet number the peer is acknowledging in this packet (typically the largest that the peer has seen thus far.)

ACK Delay: The time from when the largest acknowledged packet, as indicated in the Largest Acknowledged field, was received by this peer to when this ACK was sent.

ACK Block Section: Contains one or more blocks of packet numbers which have been successfully received, see Section 8.2.1.

Timestamp Section: Contains zero or more timestamps reporting transit delay of received packets. See Section 8.2.2.

8.2.1. ACK Block Section

The ACK Block Section contains between one and 256 blocks of packet numbers which have been successfully received. If the Num Blocks field is absent, only the First ACK Block length is present in this section. Otherwise, the Num Blocks field indicates how many additional blocks follow the First ACK Block Length field.

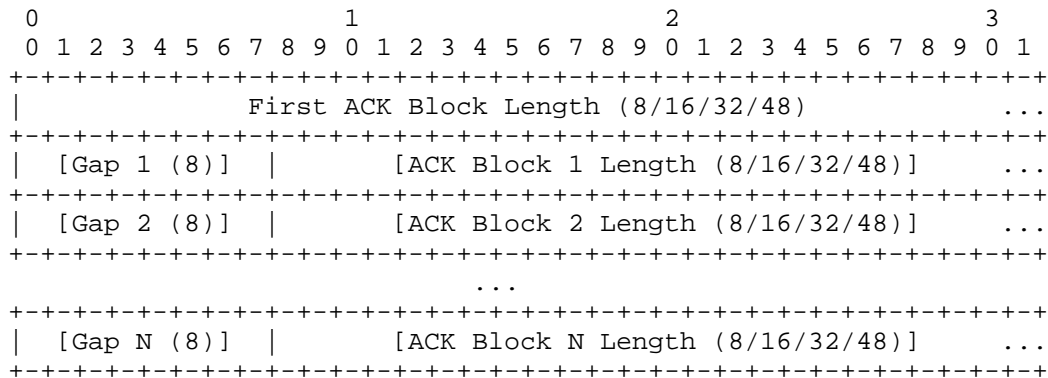


Figure 9: ACK Block Section

The fields in the ACK Block Section are:

First ACK Block Length: An unsigned packet number delta that indicates the number of contiguous additional packets being acknowledged starting at the Largest Acknowledged.

Gap To Next Block (opt, repeated): An unsigned number specifying the number of contiguous missing packets from the end of the previous ACK block to the start of the next. Repeated "Num Blocks" times.

ACK Block Length (opt, repeated): An unsigned packet number delta that indicates the number of contiguous packets being acknowledged starting after the end of the previous gap. Repeated "Num Blocks" times.

8.2.2. Timestamp Section

The Timestamp Section contains between zero and 255 measurements of packet receive times relative to the beginning of the connection.

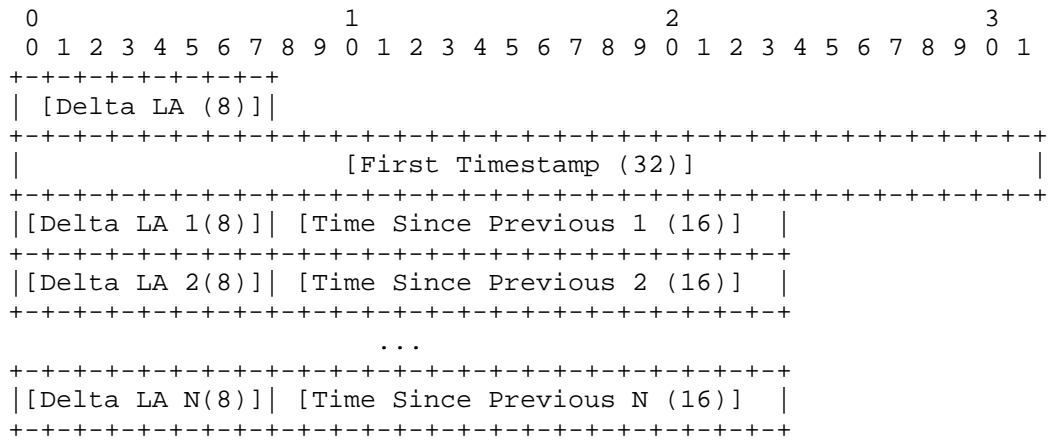


Figure 10: Timestamp Section

The fields in the Timestamp Section are:

Delta Largest Acknowledged (opt): An optional 8-bit unsigned packet number delta specifying the delta between the largest acknowledged and the first packet whose timestamp is being reported. In other words, this first packet number may be computed as (Largest Acknowledged - Delta Largest Acknowledged.)

First Timestamp (opt): An optional 32-bit unsigned value specifying the time delta in microseconds, from the beginning of the connection to the arrival of the packet indicated by Delta Largest Acknowledged.

Delta Largest Aced 1..N (opt, repeated): This field has the same semantics and format as "Delta Largest Acknowledged". Repeated "Num Timestamps - 1" times.

Time Since Previous Timestamp 1..N(opt, repeated): An optional 16-bit unsigned value specifying time delta from the previous reported timestamp. It is encoded in the same format as the ACK Delay. Repeated "Num Timestamps - 1" times.

The timestamp section lists packet receipt timestamps ordered by timestamp.

8.2.2.1. Time Format

DISCUSS_AND_REPLACE: Perhaps make this format simpler.

The time format used in the ACK frame above is a 16-bit unsigned float with 11 explicit bits of mantissa and 5 bits of explicit exponent, specifying time in microseconds. The bit format is loosely modeled after IEEE 754. For example, 1 microsecond is represented as 0x1, which has an exponent of zero, presented in the 5 high order bits, and mantissa of 1, presented in the 11 low order bits. When the explicit exponent is greater than zero, an implicit high-order 12th bit of 1 is assumed in the mantissa. For example, a floating value of 0x800 has an explicit exponent of 1, as well as an explicit mantissa of 0, but then has an effective mantissa of 4096 (12th bit is assumed to be 1). Additionally, the actual exponent is one-less than the explicit exponent, and the value represents 4096 microseconds. Any values larger than the representable range are clamped to 0xFFFF.

8.2.3. ACK Frames and Packet Protection

ACK frames that acknowledge protected packets MUST be carried in a packet that has an equivalent or greater level of packet protection.

Packets that are protected with 1-RTT keys MUST be acknowledged in packets that are also protected with 1-RTT keys.

A packet that is not protected and claims to acknowledge a packet number that was sent with packet protection is not valid. An unprotected packet that carries acknowledgments for protected packets MUST be discarded in its entirety.

Packets that a client sends with 0-RTT packet protection MUST be acknowledged by the server in packets protected by 1-RTT keys. This can mean that the client is unable to use these acknowledgments if the server cryptographic handshake messages are delayed or lost. Note that the same limitation applies to other data sent by the server protected by the 1-RTT keys.

Unprotected packets, such as those that carry the initial cryptographic handshake messages, MAY be acknowledged in unprotected packets. Unprotected packets are vulnerable to falsification or modification. Unprotected packets can be acknowledged along with protected packets in a protected packet.

An endpoint SHOULD acknowledge packets containing cryptographic handshake messages in the next unprotected packet that it sends, unless it is able to acknowledge those packets in later packets protected by 1-RTT keys. At the completion of the cryptographic handshake, both peers send unprotected packets containing cryptographic handshake messages followed by packets protected by 1-RTT keys. An endpoint SHOULD acknowledge the unprotected packets

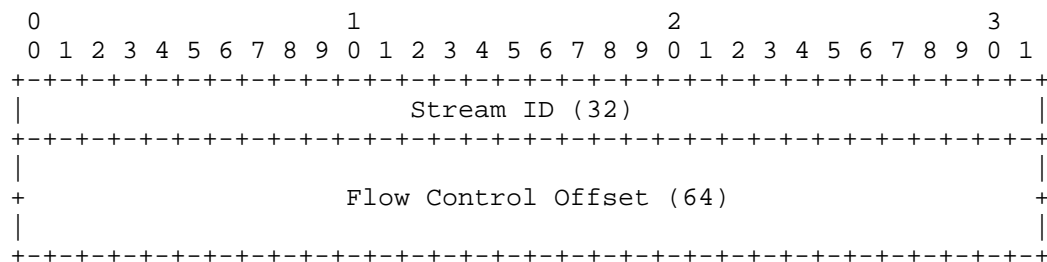
that complete the cryptographic handshake in a protected packet, because its peer is guaranteed to have access to 1-RTT packet protection keys.

For instance, a server acknowledges a TLS ClientHello in the packet that carries the TLS ServerHello; similarly, a client can acknowledge a TLS HelloRetryRequest in the packet containing a second TLS ClientHello. The complete set of server handshake messages (TLS ServerHello through to Finished) might be acknowledged by a client in protected packets, because it is certain that the server is able to decipher the packet.

8.3. WINDOW_UPDATE Frame

The WINDOW_UPDATE frame (type=0x04) informs the peer of an increase in an endpoint's flow control receive window for either a single stream, or the entire connection as a whole.

The frame is as follows:



The fields in the WINDOW_UPDATE frame are as follows:

Stream ID: ID of the stream whose flow control windows is being updated, or 0 to specify the connection-level flow control window.

Flow Control Offset: A 64-bit unsigned integer indicating the flow control offset for the given stream (for a stream ID other than 0) or the entire connection.

The flow control offset is expressed in units of octets for individual streams (for stream identifiers other than 0).

The connection-level flow control offset is expressed in units of 1024 octets (for a stream identifier of 0). That is, the connection-level flow control offset is determined by multiplying the encoded value by 1024.

An endpoint accounts for the maximum offset of data that is sent or received on a stream. Loss or reordering can mean that the maximum offset is greater than the total size of data received on a stream. Similarly, receiving STREAM frames might not increase the maximum offset on a stream. A STREAM frame with a FIN bit set or RST_STREAM causes the final offset for a stream to be fixed.

The maximum data offset on a stream MUST NOT exceed the stream flow control offset advertised by the receiver. The sum of the maximum data offsets of all streams (including closed streams) MUST NOT exceed the connection flow control offset advertised by the receiver. An endpoint MUST terminate a connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error if it receives more data than the largest flow control offset that it has sent, unless this is a result of a change in the initial offsets (see Section 7.3.2).

8.4. BLOCKED Frame

A sender sends a BLOCKED frame (type=0x05) when it is ready to send data (and has data to send), but is currently flow control blocked. BLOCKED frames are purely informational frames, but extremely useful for debugging purposes. A receiver of a BLOCKED frame should simply discard it (after possibly printing a helpful log message). The frame is as follows:

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Stream ID (32)                               |
+-----+-----+-----+-----+-----+-----+-----+-----+

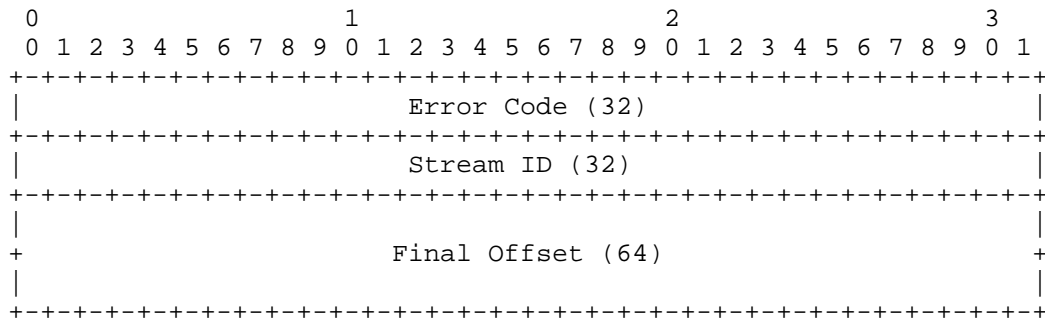
```

The BLOCKED frame contains a single field:

Stream ID: A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked.

8.5. RST_STREAM Frame

An endpoint may use a RST_STREAM frame (type=0x01) to abruptly terminate a stream. The frame is as follows:



The fields are:

Error code: A 32-bit error code which indicates why the stream is being closed.

Stream ID: The 32-bit Stream ID of the stream being terminated.

Final offset: A 64-bit unsigned integer indicating the absolute byte offset of the end of data written on this stream by the RST_STREAM sender.

8.6. PADDING Frame

The PADDING frame (type=0x00) has no semantic value. PADDING frames can be used to increase the size of a packet. Padding can be used to increase an initial client packet to the minimum required size, or to provide protection against traffic analysis for protected packets.

A PADDING frame has no content. That is, a PADDING frame consists of the single octet that identifies the frame as a PADDING frame.

8.7. PING frame

Endpoints can use PING frames (type=0x07) to verify that their peers are still alive or to check reachability to the peer. The PING frame contains no additional fields. The receiver of a PING frame simply needs to acknowledge the packet containing this frame. The PING frame SHOULD be used to keep a connection alive when a stream is open. The default is to send a PING frame after 15 seconds of quiescence. A PING frame has no additional fields.

8.8. CONNECTION_CLOSE frame

An endpoint sends a CONNECTION_CLOSE frame (type=0x02) to notify its peer that the connection is being closed. If there are open streams that haven't been explicitly closed, they are implicitly closed when

the connection is closed. (Ideally, a GOAWAY frame would be sent with enough time that all streams are torn down.) The frame is as follows:

```

      0               1               2               3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Error Code (32)                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Reason Phrase Length (16) | [Reason Phrase (*)] | ... |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

The fields of a CONNECTION_CLOSE frame are as follows:

Error Code: A 32-bit error code which indicates the reason for closing this connection.

Reason Phrase Length: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the Error Code.

Reason Phrase: An optional human-readable explanation for why the connection was closed.

8.9. GOAWAY Frame

An endpoint uses a GOAWAY frame (type=0x03) to initiate a graceful shutdown of a connection. The endpoints will continue to use any active streams, but the sender of the GOAWAY will not initiate or accept any additional streams beyond those indicated. The GOAWAY frame is as follows:

```

      0               1               2               3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Largest Client Stream ID (32)                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Largest Server Stream ID (32)                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

The fields of a GOAWAY frame are:

Largest Client Stream ID: The highest-numbered, client-initiated stream on which the endpoint sending the GOAWAY frame either sent data, or received and delivered data. All higher-numbered, client-initiated streams (that is, odd-numbered streams) are implicitly reset by sending or receiving the GOAWAY frame.

Largest Server Stream ID: The highest-numbered, server-initiated stream on which the endpoint sending the GOAWAY frame either sent data, or received and delivered data. All higher-numbered, server-initiated streams (that is, even-numbered streams) are implicitly reset by sending or receiving the GOAWAY frame.

A GOAWAY frame indicates that any application layer actions on streams with higher numbers than those indicated can be safely retried because no data was exchanged. An endpoint **MUST** set the value of the Largest Client or Server Stream ID to be at least as high as the highest-numbered stream on which it either sent data or received and delivered data to the application protocol that uses QUIC.

An endpoint **MAY** choose a larger stream identifier if it wishes to allow for a number of streams to be created. This is especially valuable for peer-initiated streams where packets creating new streams could be in transit; using a larger stream number allows those streams to complete.

In addition to initiating a graceful shutdown of a connection, GOAWAY **MAY** be sent immediately prior to sending a CONNECTION_CLOSE frame that is sent as a result of detecting a fatal error. Higher-numbered streams than those indicated in the GOAWAY frame can then be retried.

9. Packetization and Reliability

The Path Maximum Transmission Unit (PMTU) is the maximum size of the entire IP header, UDP header, and UDP payload. The UDP payload includes the QUIC public header, encrypted payload, and any authentication fields.

All QUIC packets **SHOULD** be sized to fit within the estimated PMTU to avoid IP fragmentation or packet drops. To optimize bandwidth efficiency, endpoints **SHOULD** use Packetization Layer PMTU Discovery ([RFC4821]) and **MAY** use PMTU Discovery ([RFC1191], [RFC1981]) for detecting the PMTU, setting the PMTU appropriately, and storing the result of previous PMTU determinations.

In the absence of these mechanisms, QUIC endpoints **SHOULD NOT** send IP packets larger than 1280 octets. Assuming the minimum IP header size, this results in a UDP payload length of 1232 octets for IPv6 and 1252 octets for IPv4.

QUIC endpoints that implement any kind of PMTU discovery **SHOULD** maintain an estimate for each combination of local and remote IP addresses (as each pairing could have a different maximum MTU in the path).

QUIC depends on the network path supporting a MTU of at least 1280 octets. This is the IPv6 minimum and therefore also supported by most modern IPv4 networks. An endpoint **MUST NOT** reduce their MTU below this number, even if it receives signals that indicate a smaller limit might exist.

Clients **MUST** ensure that the first packet in a connection, and any retransmissions of those octets, has a total size (including IP and UDP headers) of at least 1280 bytes. This might require inclusion of PADDING frames. It is **RECOMMENDED** that a packet be padded to exactly 1280 octets unless the client has a reasonable assurance that the PMTU is larger. Sending a packet of this size ensures that the network path supports an MTU of this size and helps mitigate amplification attacks caused by server responses toward an unverified client address.

Servers **MUST** reject the first plaintext packet received from a client if its total size is less than 1280 octets, to mitigate amplification attacks.

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses has fallen below 1280 octets, it **MUST** immediately cease sending QUIC packets between those IP addresses. This may result in abrupt termination of the connection if all pairs are affected. In this case, an endpoint **SHOULD** send a Public Reset packet to indicate the failure. The application **SHOULD** attempt to use TLS over TCP instead.

A sender bundles one or more frames in a Regular QUIC packet (see Section 6).

A sender **SHOULD** minimize per-packet bandwidth and computational costs by bundling as many frames as possible within a QUIC packet. A sender **MAY** wait for a short period of time to bundle multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets. An implementation may use heuristics about expected application sending behavior to determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

Regular QUIC packets are "containers" of frames; a packet is never retransmitted whole. How an endpoint handles the loss of the frame depends on the type of the frame. Some frames are simply retransmitted, some have their contents moved to new frames, and others are never retransmitted.

When a packet is detected as lost, the sender re-sends any frames as necessary:

- o All application data sent in STREAM frames MUST be retransmitted, unless the endpoint has sent a RST_STREAM for that stream. When an endpoint sends a RST_STREAM frame, data outstanding on that stream SHOULD NOT be retransmitted, since subsequent data on this stream is expected to not be delivered by the receiver.
- o ACK and PADDING frames MUST NOT be retransmitted. ACK frames are cumulative, so new frames containing updated information will be sent as described in Section 8.2.
- o All other frames MUST be retransmitted.

Upon detecting losses, a sender MUST take appropriate congestion control action. The details of loss detection and congestion control are described in [QUIC-RECOVERY].

A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. For STREAM frames, this means the data has been queued (but not necessarily delivered to the application). This also means that any stream state transitions triggered by STREAM or RST_STREAM frames have occurred. Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet.

To avoid creating an indefinite feedback loop, an endpoint MUST NOT generate an ACK frame in response to a packet containing only ACK or PADDING frames.

Strategies and implications of the frequency of generating acknowledgments are discussed in more detail in [QUIC-RECOVERY].

9.1. Special Considerations for PMTU Discovery

Traditional ICMP-based path MTU discovery in IPv4 ([RFC1191] is potentially vulnerable to off-path attacks that successfully guess the IP/port 4-tuple and reduce the MTU to a bandwidth-inefficient value. TCP connections mitigate this risk by using the (at minimum) 8 bytes of transport header echoed in the ICMP message to validate the TCP sequence number as valid for the current connection. However, as QUIC operates over UDP, in IPv4 the echoed information could consist only of the IP and UDP headers, which usually has insufficient entropy to mitigate off-path attacks.

As a result, endpoints that implement PMTUD in IPv4 SHOULD take steps to mitigate this risk. For instance, an application could:

- o Set the IPv4 Don't Fragment (DF) bit on a small proportion of packets, so that most invalid ICMP messages arrive when there are no DF packets outstanding, and can therefore be identified as spurious.
- o Store additional information from the IP or UDP headers from DF packets (for example, the IP ID or UDP checksum) to further authenticate incoming Datagram Too Big messages.
- o Any reduction in PMTU due to a report contained in an ICMP packet is provisional until QUIC's loss detection algorithm determines that the packet is actually lost.

10. Streams: QUIC's Data Structuring Abstraction

Streams in QUIC provide a lightweight, ordered, and bidirectional byte-stream abstraction modeled closely on HTTP/2 streams [RFC7540].

Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled.

Data that is received on a stream is delivered in order within that stream, but there is no particular delivery order across streams. Transmit ordering among streams is left to the implementation.

The creation and destruction of streams are expected to have minimal bandwidth and computational cost. A single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection.

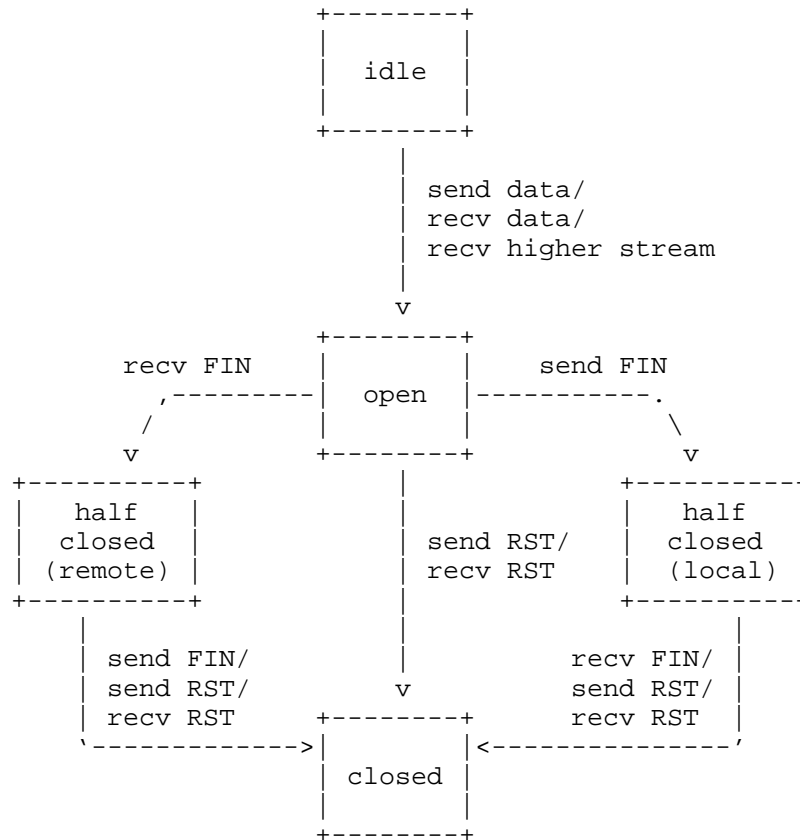
Streams are individually flow controlled, allowing an endpoint to limit memory commitment and to apply back pressure.

An alternative view of QUIC streams is as an elastic "message" abstraction, similar to the way ephemeral streams are used in SST [SST], which may be a more appealing description for some applications.

10.1. Life of a Stream

The semantics of QUIC streams is based on HTTP/2 streams, and the lifecycle of a QUIC stream therefore closely follows that of an HTTP/2 stream [RFC7540], with some differences to accommodate the possibility of out-of-order delivery due to the use of multiple

streams in QUIC. The lifecycle of a QUIC stream is shown in the following figure and described below.



send: endpoint sends this frame
recv: endpoint receives this frame

data: application data in a STREAM frame
FIN: FIN flag in a STREAM frame
RST: RST_STREAM frame

Figure 11: Lifecycle of a stream

Note that this diagram shows stream state transitions and the frames and flags that affect those transitions only. For the purpose of state transitions, the FIN flag is processed as a separate event to the frame that bears it; a STREAM frame with the FIN flag set can cause two state transitions. When the FIN flag is sent on an empty

STREAM frame, the offset in the STREAM frame MUST be one greater than the last data byte sent on this stream.

The recipient of a frame which changes stream state will have a delayed view of the state of a stream while the frame is in transit. Endpoints do not coordinate the creation of streams; they are created unilaterally by either endpoint. The negative consequences of a mismatch in states are limited to the "closed" state after sending RST_STREAM, where frames might be received for some time after closing. Endpoints can use acknowledgments to understand the peer's subjective view of stream state at any given time.

Streams have the following states:

10.1.1.1. idle

All streams start in the "idle" state.

The following transitions are valid from this state:

Sending or receiving a STREAM frame causes the stream to become "open". The stream identifier is selected as described in Section 10.2. The same STREAM frame can also cause a stream to immediately become "half-closed".

Receiving a STREAM frame on a peer-initiated stream (that is, a packet sent by a server on an even-numbered stream or a client packet on an odd-numbered stream) also causes all lower-numbered "idle" streams in the same direction to become "open". This could occur if a peer begins sending on streams in a different order to their creation, or it could happen if packets are lost or reordered in transit.

Receiving any frame other than STREAM or RST_STREAM on a stream in this state MUST be treated as a connection error (Section 12) of type YYY.

10.1.1.2. open

A stream in the "open" state may be used by both peers to send frames of any type. In this state, a sending peer must observe the flow-control limit advertised by its receiving peer (Section 11).

From this state, either endpoint can send a frame with the FIN flag set, which causes the stream to transition into one of the "half-closed" states. An endpoint sending an FIN flag causes the stream state to become "half-closed (local)". An endpoint receiving a FIN flag causes the stream state to become "half-closed (remote)" once

all preceding data has arrived. The receiving endpoint MUST NOT consider the stream state to have changed until all data has arrived.

Either endpoint can send a RST_STREAM frame from this state, causing it to transition immediately to "closed".

10.1.3. half-closed (local)

A stream that is in the "half-closed (local)" state MUST NOT be used for sending STREAM frames; WINDOW_UPDATE and RST_STREAM MAY be sent in this state.

A stream transitions from this state to "closed" when a STREAM frame that contains a FIN flag is received and all prior data has arrived, or when either peer sends a RST_STREAM frame.

An endpoint that closes a stream MUST NOT send data beyond the final offset that it has chosen, see Section 10.1.5 for details.

An endpoint can receive any type of frame in this state. Providing flow-control credit using WINDOW_UPDATE frames is necessary to continue receiving flow-controlled frames. In this state, a receiver MAY ignore WINDOW_UPDATE frames for this stream, which might arrive for a short period after a frame bearing the FIN flag is sent.

10.1.4. half-closed (remote)

A stream that is "half-closed (remote)" is no longer being used by the peer to send any data. In this state, a sender is no longer obligated to maintain a receiver stream-level flow-control window.

A stream that is in the "half-closed (remote)" state will have a final offset for received data, see Section 10.1.5 for details.

A stream in this state can be used by the endpoint to send frames of any type. In this state, the endpoint continues to observe advertised stream-level and connection-level flow-control limits (Section 11).

A stream can transition from this state to "closed" by sending a frame that contains a FIN flag or when either peer sends a RST_STREAM frame.

10.1.5. closed

The "closed" state is the terminal state.

An endpoint will learn the final offset of the data it receives on a stream when it enters the "half-closed (remote)" or "closed" state. The final offset is carried explicitly in the RST_STREAM frame; otherwise, the final offset is the offset of the end of the data carried in STREAM frame marked with a FIN flag.

An endpoint MUST NOT send data on a stream at or beyond the final offset.

Once a final offset for a stream is known, it cannot change. If a RST_STREAM or STREAM frame causes the final offset to change for a stream, an endpoint SHOULD respond with a QUIC_STREAM_DATA_AFTER_TERMINATION error (see Section 12). A receiver SHOULD treat receipt of data at or beyond the final offset as a QUIC_STREAM_DATA_AFTER_TERMINATION error. Generating these errors is not mandatory, but only because requiring that an endpoint generate these errors also means that the endpoint needs to maintain the final offset state for closed streams, which could mean a significant state commitment.

An endpoint that receives a RST_STREAM frame (and which has not sent a FIN or a RST_STREAM) MUST immediately respond with a RST_STREAM frame, and MUST NOT send any more data on the stream. This endpoint may continue receiving frames for the stream on which a RST_STREAM is received.

If this state is reached as a result of sending a RST_STREAM frame, the peer that receives the RST_STREAM frame might have already sent - or enqueued for sending - frames on the stream that cannot be withdrawn. An endpoint MUST ignore frames that it receives on closed streams after it has sent a RST_STREAM frame. An endpoint MAY choose to limit the period over which it ignores frames and treat frames that arrive after this time as being in error.

STREAM frames received after sending RST_STREAM are counted toward the connection and stream flow-control windows. Even though these frames might be ignored, because they are sent before their sender receives the RST_STREAM, the sender will consider the frames to count against its flow-control windows.

In the absence of more specific guidance elsewhere in this document, implementations SHOULD treat the receipt of a frame that is not expressly permitted in the description of a state as a connection error (Section 12). Frames of unknown types are ignored.

(TODO: QUIC_STREAM_NO_ERROR is a special case. Write it up.)

10.2. Stream Identifiers

Streams are identified by an unsigned 32-bit integer, referred to as the StreamID. To avoid StreamID collision, clients MUST initiate streams using odd-numbered StreamIDs; streams initiated by the server MUST use even-numbered StreamIDs.

A StreamID of zero (0x0) is reserved and used for connection-level flow control frames (Section 11); the StreamID of zero cannot be used to establish a new stream.

StreamID 1 (0x1) is reserved for the cryptographic handshake. StreamID 1 MUST NOT be used for application data, and MUST be the first client-initiated stream.

A QUIC endpoint cannot reuse a StreamID on a given connection. Streams MUST be created in sequential order. Open streams can be used in any order. Streams that are used out of order result in lower-numbered streams in the same direction being counted as open.

All streams, including stream 1, count toward this limit. Thus, a concurrent stream limit of 0 will cause a connection to be unusable. Application protocols that use QUIC might require a certain minimum number of streams to function correctly. If a peer advertises a concurrent stream limit (concurrent_streams) that is too small for the selected application protocol to function, an endpoint MUST terminate the connection with an error of type QUIC_TOO_MANY_OPEN_STREAMS (Section 12).

10.3. Stream Concurrency

An endpoint limits the number of concurrently active incoming streams by setting the concurrent stream limit (see Section 7.3.1) in the transport parameters. The maximum concurrent streams setting is specific to each endpoint and applies only to the peer that receives the setting. That is, clients specify the maximum number of concurrent streams the server can initiate, and servers specify the maximum number of concurrent streams the client can initiate.

Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the concurrent stream limit.

A recently closed stream MUST also be considered to count toward this limit until packets containing all frames required to close the stream have been acknowledged. For a stream which closed cleanly, this means all STREAM frames have been acknowledged; for a stream

which closed abruptly, this means the RST_STREAM frame has been acknowledged.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a STREAM frame that causes its advertised concurrent stream limit to be exceeded MUST treat this as a stream error of type QUIC_TOO_MANY_OPEN_STREAMS (Section 12).

10.4. Sending and Receiving Data

Once a stream is created, endpoints may use the stream to send and receive data. Each endpoint may send a series of STREAM frames encapsulating data on a stream until the stream is terminated in that direction. Streams are an ordered byte-stream abstraction, and they have no other structure within them. STREAM frame boundaries are not expected to be preserved in retransmissions from the sender or during delivery to the application at the receiver.

When new data is to be sent on a stream, a sender MUST set the encapsulating STREAM frame's offset field to the stream offset of the first byte of this new data. The first byte of data that is sent on a stream has the stream offset 0. The largest offset delivered on a stream MUST be less than 2^{64} . A receiver MUST ensure that received stream data is delivered to the application as an ordered byte-stream. Data received out of order MUST be buffered for later delivery, as long as it is not in violation of the receiver's flow control limits.

The cryptographic handshake stream, Stream 1, MUST NOT be subject to congestion control or connection-level flow control, but MUST be subject to stream-level flow control. An endpoint MUST NOT send data on any other stream without consulting the congestion controller and the flow controller.

Flow control is described in detail in Section 11, and congestion control is described in the companion document [QUIC-RECOVERY].

10.5. Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2 [RFC7540], shows that effective prioritization strategies have a significant positive impact on performance.

QUIC does not provide frames for exchanging prioritization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to

define any prioritization scheme that suits their application semantics. A protocol might define explicit messages for signaling priority, such as those defined in HTTP/2; it could define rules that allow an endpoint to determine priority based on context; or it could leave the determination to the application.

A QUIC implementation SHOULD provide ways in which an application can indicate the relative priority of streams. When deciding which streams to dedicate resources to, QUIC SHOULD use the information provided by the application. Failure to account for priority of streams can result in suboptimal performance.

Stream priority is most relevant when deciding which stream data will be transmitted. Often, there will be limits on what can be transmitted as a result of connection flow control or the current congestion controller state.

Giving preference to the transmission of its own management frames ensures that the protocol functions efficiently. That is, prioritizing frames other than STREAM frames ensures that loss recovery, congestion control, and flow control operate effectively.

Stream 1 MUST be prioritized over other streams prior to the completion of the cryptographic handshake. This includes the retransmission of the second flight of client handshake messages, that is, the TLS Finished and any client authentication messages.

STREAM frames that are determined to be lost SHOULD be retransmitted before sending new data, unless application priorities indicate otherwise. Retransmitting lost STREAM frames can fill in gaps, which allows the peer to consume already received data and free up flow control window.

11. Flow Control

It is necessary to limit the amount of data that a sender may have outstanding at any time, so as to prevent a fast sender from overwhelming a slow receiver, or to prevent a malicious sender from consuming significant resources at a receiver. This section describes QUIC's flow-control mechanisms.

QUIC employs a credit-based flow-control scheme similar to HTTP/2's flow control [RFC7540]. A receiver advertises the number of octets it is prepared to receive on a given stream and for the entire connection. This leads to two levels of flow control in QUIC: (i) Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, and (ii) Stream flow

control, which prevents a single stream from consuming the entire receive buffer for a connection.

A receiver sends WINDOW_UPDATE frames to the sender to advertise additional credit by sending the absolute byte offset in the stream or in the connection which it is willing to receive.

The initial flow control credit is 65536 bytes for both the stream and connection flow controllers.

A receiver MAY advertise a larger offset at any point in the connection by sending a WINDOW_UPDATE frame. A receiver MUST NOT renege on an advertisement; that is, once a receiver advertises an offset via a WINDOW_UPDATE frame, it MUST NOT subsequently advertise a smaller offset. A sender may receive WINDOW_UPDATE frames out of order; a sender MUST therefore ignore any WINDOW_UPDATE that does not move the window forward.

A receiver MUST close the connection with a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA error (Section 12) if the peer violates the advertised stream or connection flow control windows.

A sender MUST send BLOCKED frames to indicate it has data to write but is blocked by lack of connection or stream flow control credit. BLOCKED frames are expected to be sent infrequently in common cases, but they are considered useful for debugging and monitoring purposes.

A receiver advertises credit for a stream by sending a WINDOW_UPDATE frame with the StreamID set appropriately. A receiver may use the current offset of data consumed to determine the flow control offset to be advertised. A receiver MAY send copies of a WINDOW_UPDATE frame in multiple packets in order to make sure that the sender receives it before running out of flow control credit, even if one of the packets is lost.

Connection flow control is a limit to the total bytes of stream data sent in STREAM frames on all streams contributing to connection flow control. A receiver advertises credit for a connection by sending a WINDOW_UPDATE frame with the StreamID set to zero (0x00). A receiver maintains a cumulative sum of bytes received on all streams contributing to connection-level flow control, to check for flow control violations. A receiver may maintain a cumulative sum of bytes consumed on all contributing streams to determine the connection-level flow control offset to be advertised.

11.1. Edge Cases and Other Considerations

There are some edge cases which must be considered when dealing with stream and connection level flow control. Given enough time, both endpoints must agree on flow control state. If one end believes it can send more than the other end is willing to receive, the connection will be torn down when too much data arrives. Conversely if a sender believes it is blocked, while endpoint B expects more data can be received, then the connection can be in a deadlock, with the sender waiting for a WINDOW_UPDATE which will never come.

11.1.1. Mid-stream RST_STREAM

On receipt of a RST_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream. This could result in the endpoints getting out of sync, since the RST_STREAM frame may have arrived out of order and there may be further bytes in flight. The data sender would have counted the data against its connection level flow control budget, but a receiver that has not received these bytes would not know to include them as well. The receiver must learn the number of bytes that were sent on the stream to make the same adjustment in its connection flow controller.

To avoid this de-synchronization, a RST_STREAM sender MUST include the final byte offset sent on the stream in the RST_STREAM frame. On receiving a RST_STREAM frame, a receiver definitively knows how many bytes were sent on that stream before the RST_STREAM frame, and the receiver MUST use the final offset to account for all bytes sent on the stream in its connection level flow controller.

11.1.2. Response to a RST_STREAM

Since streams are bidirectional, a sender of a RST_STREAM needs to know how many bytes the peer has sent on the stream. If an endpoint receives a RST_STREAM frame and has sent neither a FIN nor a RST_STREAM, it MUST send a RST_STREAM in response, bearing the offset of the last byte sent on this stream as the final offset.

11.1.3. Offset Increment

This document leaves when and how many bytes to advertise in a WINDOW_UPDATE to the implementation, but offers a few considerations. WINDOW_UPDATE frames constitute overhead, and therefore, sending a WINDOW_UPDATE with small offset increments is undesirable. At the same time, sending WINDOW_UPDATES with large offset increments requires the sender to commit to that amount of buffer.

Implementations must find the correct tradeoff between these sides to determine how large an offset increment to send in a WINDOW_UPDATE.

A receiver MAY use an autotuning mechanism to tune the size of the offset increment to advertise based on a roundtrip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations.

11.1.4. BLOCKED frames

If a sender does not receive a WINDOW_UPDATE frame when it has run out of flow control credit, the sender will be blocked and MUST send a BLOCKED frame. A BLOCKED frame is expected to be useful for debugging at the receiver. A receiver SHOULD NOT wait for a BLOCKED frame before sending a WINDOW_UPDATE, since doing so will cause at least one roundtrip of quiescence. For smooth operation of the congestion controller, it is generally considered best to not let the sender go into quiescence if avoidable. To avoid blocking a sender, and to reasonably account for the possibility of loss, a receiver should send a WINDOW_UPDATE frame at least two roundtrips before it expects the sender to get blocked.

12. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Errors can affect an entire connection (see Section 12.1), or a single stream (see Section 12.2).

The most appropriate error code (Section 12.3) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used.

Public Reset is not suitable for any error that can be signaled with a CONNECTION_CLOSE or RST_STREAM frame. Public Reset MUST NOT be sent by an endpoint that has the state necessary to send a frame on the connection.

12.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a CONNECTION_CLOSE frame (Section 8.8). An endpoint MAY close the connection in this manner, even if the error only affects a single stream.

A `CONNECTION_CLOSE` frame could be sent in a packet that is lost. An endpoint **SHOULD** be prepared to retransmit a packet containing a `CONNECTION_CLOSE` frame if it receives more packets on a terminated connection. Limiting the number of retransmissions and the time over which this final packet is sent limits the effort expended on terminated connections.

An endpoint that chooses not to retransmit packets containing `CONNECTION_CLOSE` risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to send a Public Reset packet.

12.2. Stream Errors

If the error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can send a `RST_STREAM` frame (Section 8.5) with an appropriate error code to terminate just the affected stream.

Stream 1 is critical to the functioning of the entire connection. If stream 1 is closed with either a `RST_STREAM` or `STREAM` frame bearing the `FIN` flag, an endpoint **MUST** generate a connection error of type `QUIC_CLOSED_CRITICAL_STREAM`.

Some application protocols make other streams critical to that protocol. An application protocol does not need to inform the transport that a stream is critical; it can instead generate appropriate errors in response to being notified that the critical stream is closed.

An endpoint **MAY** send a `RST_STREAM` frame in the same packet as a `CONNECTION_CLOSE` frame.

12.3. Error Codes

Error codes are 32 bits long, with the first two bits indicating the source of the error code:

`0x00000000-0x3FFFFFFF`: Application-specific error codes. Defined by each application-layer protocol.

`0x40000000-0x7FFFFFFF`: Reserved for host-local error codes. These codes **MUST NOT** be sent to a peer, but **MAY** be used in API return codes and logs.

`0x80000000-0xBFFFFFFF`: QUIC transport error codes, including packet protection errors. Applicable to all uses of QUIC.

0xC0000000-0xFFFFFFFF: Cryptographic error codes. Defined by the cryptographic handshake protocol in use.

This section lists the defined QUIC transport error codes that may be used in a CONNECTION_CLOSE or RST_STREAM frame. Error codes share a common code space. Some error codes apply only to either streams or the entire connection and have no defined semantics in the other context.

QUIC_INTERNAL_ERROR (0x80000001): Connection has reached an invalid state.

QUIC_STREAM_DATA_AFTER_TERMINATION (0x80000002): There were data frames after the a fin or reset.

QUIC_INVALID_PACKET_HEADER (0x80000003): Control frame is malformed.

QUIC_INVALID_FRAME_DATA (0x80000004): Frame data is malformed.

QUIC_MULTIPLE_TERMINATION_OFFSETS (0x80000005): Multiple final offset values were received on the same stream

QUIC_STREAM_CANCELLED (0x80000006): The stream was cancelled

QUIC_CLOSED_CRITICAL_STREAM (0x80000007): A stream that is critical to the protocol was closed.

QUIC_MISSING_PAYLOAD (0x80000030): The packet contained no payload.

QUIC_INVALID_STREAM_DATA (0x8000002E): STREAM frame data is malformed.

QUIC_UNENCRYPTED_STREAM_DATA (0x8000003D): Received STREAM frame data is not encrypted.

QUIC_MAYBE_CORRUPTED_MEMORY (0x80000059): Received a frame which is likely the result of memory corruption.

QUIC_INVALID_RST_STREAM_DATA (0x80000006): RST_STREAM frame data is malformed.

QUIC_INVALID_CONNECTION_CLOSE_DATA (0x80000007): CONNECTION_CLOSE frame data is malformed.

QUIC_INVALID_GOAWAY_DATA (0x80000008): GOAWAY frame data is malformed.

QUIC_INVALID_WINDOW_UPDATE_DATA (0x80000039): WINDOW_UPDATE frame data is malformed.

QUIC_INVALID_BLOCKED_DATA (0x8000003A): BLOCKED frame data is malformed.

QUIC_INVALID_PATH_CLOSE_DATA (0x8000004E): PATH_CLOSE frame data is malformed.

QUIC_INVALID_ACK_DATA (0x80000009): ACK frame data is malformed.

QUIC_INVALID_VERSION_NEGOTIATION_PACKET (0x8000000A): Version negotiation packet is malformed.

QUIC_INVALID_PUBLIC_RST_PACKET (0x8000000b): Public RST packet is malformed.

QUIC_DECRYPTION_FAILURE (0x8000000c): There was an error decrypting.

QUIC_ENCRYPTION_FAILURE (0x8000000d): There was an error encrypting.

QUIC_PACKET_TOO_LARGE (0x8000000e): The packet exceeded kMaxPacketSize.

QUIC_PEER_GOING_AWAY (0x80000010): The peer is going away. May be a client or server.

QUIC_INVALID_STREAM_ID (0x80000011): A stream ID was invalid.

QUIC_INVALID_PRIORITY (0x80000031): A priority was invalid.

QUIC_TOO_MANY_OPEN_STREAMS (0x80000012): Too many streams already open.

QUIC_TOO_MANY_AVAILABLE_STREAMS (0x8000004c): The peer created too many available streams.

QUIC_PUBLIC_RESET (0x80000013): Received public reset for this connection.

QUIC_INVALID_VERSION (0x80000014): Invalid protocol version.

QUIC_INVALID_HEADER_ID (0x80000016): The Header ID for a stream was too far from the previous.

QUIC_INVALID_NEGOTIATED_VALUE (0x80000017): Negotiable parameter received during handshake had invalid value.

QUIC_DECOMPRESSION_FAILURE (0x80000018): There was an error decompressing data.

QUIC_NETWORK_IDLE_TIMEOUT (0x80000019): The connection timed out due to no network activity.

QUIC_HANDSHAKE_TIMEOUT (0x80000043): The connection timed out waiting for the handshake to complete.

QUIC_ERROR_MIGRATING_ADDRESS (0x8000001a): There was an error encountered migrating addresses.

QUIC_ERROR_MIGRATING_PORT (0x80000056): There was an error encountered migrating port only.

QUIC_EMPTY_STREAM_FRAME_NO_FIN (0x80000032): We received a STREAM_FRAME with no data and no fin flag set.

QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA (0x8000003b): The peer received too much data, violating flow control.

QUIC_FLOW_CONTROL_SENT_TOO_MUCH_DATA (0x8000003f): The peer sent too much data, violating flow control.

QUIC_FLOW_CONTROL_INVALID_WINDOW (0x80000040): The peer received an invalid flow control window.

QUIC_CONNECTION_IP_POOLED (0x8000003e): The connection has been IP pooled into an existing connection.

QUIC_TOO_MANY_OUTSTANDING_SENT_PACKETS (0x80000044): The connection has too many outstanding sent packets.

QUIC_TOO_MANY_OUTSTANDING_RECEIVED_PACKETS (0x80000045): The connection has too many outstanding received packets.

QUIC_CONNECTION_CANCELLED (0x80000046): The QUIC connection has been cancelled.

QUIC_BAD_PACKET_LOSS_RATE (0x80000047): Disabled QUIC because of high packet loss rate.

QUIC_PUBLIC_RESETS_POST_HANDSHAKE (0x80000049): Disabled QUIC because of too many PUBLIC_RESETs post handshake.

QUIC_TIMEOUTS_WITH_OPEN_STREAMS (0x8000004a): Disabled QUIC because of too many timeouts with streams open.

QUIC_TOO_MANY_RTOS (0x80000055): QUIC timed out after too many RTOs.

QUIC_ENCRYPTION_LEVEL_INCORRECT (0x8000002c): A packet was received with the wrong encryption level (i.e. it should have been encrypted but was not.)

QUIC_VERSION_NEGOTIATION_MISMATCH (0x80000037): This connection involved a version negotiation which appears to have been tampered with.

QUIC_IP_ADDRESS_CHANGED (0x80000050): IP address changed causing connection close.

QUIC_ADDRESS_VALIDATION_FAILURE (0x80000051): Client address validation failed.

QUIC_TOO_MANY_FRAME_GAPS (0x8000005d): Stream frames arrived too discontinuously so that stream sequencer buffer maintains too many gaps.

QUIC_TOO_MANY_SESSIONS_ON_SERVER (0x80000060): Connection closed because server hit max number of sessions allowed.

13. Security and Privacy Considerations

13.1. Spoofed ACK Attack

An attacker receives an STK from the server and then releases the IP address on which it received the STK. The attacker may, in the future, spoof this same address (which now presumably addresses a different endpoint), and initiate a 0-RTT connection with a server on the victim's behalf. The attacker then spoofs ACK frames to the server which cause the server to potentially drown the victim in data.

There are two possible mitigations to this attack. The simplest one is that a server can unilaterally create a gap in packet-number space. In the non-attack scenario, the client will send an ACK frame with the larger value for largest acknowledged. In the attack scenario, the attacker could acknowledge a packet in the gap. If the server sees an acknowledgment for a packet that was never sent, the connection can be aborted.

The second mitigation is that the server can require that acknowledgments for sent packets match the encryption level of the sent packet. This mitigation is useful if the connection has an ephemeral forward-secure key that is generated and used for every new connection. If a packet sent is encrypted with a forward-secure key,

then any acknowledgments that are received for them MUST also be forward-secure encrypted. Since the attacker will not have the forward secure key, the attacker will not be able to generate forward-secure encrypted packets with ACK frames.

14. IANA Considerations

14.1. QUIC Transport Parameter Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Parameters" under a "QUIC Protocol" heading.

The "QUIC Transport Parameters" registry governs a 16-bit space. This space is split into two spaces that are governed by different policies. Values with the first byte in the range 0x00 to 0xfe (in hexadecimal) are assigned via the Specification Required policy [RFC5226]. Values with the first byte 0xff are reserved for Private Use [RFC5226].

Registrations MUST include the following fields:

Value: The numeric value of the assignment (registrations will be between 0x0000 and 0xfeff).

Parameter Name: A short mnemonic for the parameter.

Specification: A reference to a publicly available specification for the value.

The nominated expert(s) verify that a specification exists and is readily accessible. The expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious).

The initial contents of this registry are shown in Table 4.

Value	Parameter Name	Specification
0x0000	stream_fc_offset	Section 7.3.1
0x0001	connection_fc_offset	Section 7.3.1
0x0002	concurrent_streams	Section 7.3.1
0x0003	idle_timeout	Section 7.3.1
0x0004	truncate_connection_id	Section 7.3.1

Table 4: Initial QUIC Transport Parameters Entries

15. References

15.1. Normative References

- [I-D.ietf-tls-tls13]
 Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-19 (work in progress), March 2017.
- [QUIC-RECOVERY]
 Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control".
- [QUIC-TLS]
 Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC".
- [RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <<http://www.rfc-editor.org/info/rfc1191>>.
- [RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <<http://www.rfc-editor.org/info/rfc1981>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

- [RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <<http://www.rfc-editor.org/info/rfc4821>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.

15.2. Informative References

- [EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", December 2013, <<https://goo.gl/dMVtFi>>.
- [RFC2360] Scott, G., "Guide for Internet Standards Writers", BCP 22, RFC 2360, DOI 10.17487/RFC2360, June 1998, <<http://www.rfc-editor.org/info/rfc2360>>.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<http://www.rfc-editor.org/info/rfc4086>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<http://www.rfc-editor.org/info/rfc6824>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<http://www.rfc-editor.org/info/rfc7540>>.
- [SST] Ford, B., "Structured Streams: A New Transport Abstraction", DOI 10.1145/1282427.1282421, ACM SIGCOMM Computer Communication Review Volume 37 Issue 4, October 2007.

15.3. URIs

[1] <https://github.com/quicwg/base-drafts/wiki/QUIC-Versions>

Appendix A. Contributors

The original authors of this specification were Ryan Hamilton, Jana Iyengar, Ian Swett, and Alyssa Wilk.

The original design and rationale behind this protocol draw significantly from work by Jim Roskind [EARLY-DESIGN]. In alphabetical order, the contributors to the pre-IETF QUIC project at Google are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

Appendix B. Acknowledgments

Special thanks are due to the following for helping shape pre-IETF QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund.

This document has benefited immensely from various private discussions and public ones on the quic@ietf.org and proto-quic@chromium.org mailing lists. Our thanks to all.

Appendix C. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

C.1. Since draft-ietf-quic-transport-01:

- o Defined short and long packet headers (#40, #148, #361)
- o Defined a versioning scheme and stable fields (#51, #361)
- o Define reserved version values for "greasing" negotiation (#112, #278)
- o The initial packet number is randomized (#35, #283)
- o Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)

- o Defined client address validation (#52, #118, #120, #275)
- o Define transport parameters as a TLS extension (#122)
- o SCUP and COPT parameters are no longer valid (#116, #117)
- o Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- o The server chooses connection IDs in its final flight (#119, #349, #361)
- o The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- o Defined a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- o Path MTU Discovery (#64, #106)
- o The initial handshake packet from the client needs to fit in a single packet (#338)
- o Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- o Require that frames are processed when packets are acknowledged (#381, #341)
- o Removed the STOP_WAITING frame (#66)
- o Don't require retransmission of old timestamps for lost ACK frames (#308)
- o Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- o Error handling definitions (#335)
- o Split error codes into four sections (#74)
- o Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- o Define packet protection rules (#336)

- o Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RST_STREAM, before it closes (#381)
- o Remove stream reservation from state machine (#174, #280)
- o Only stream 0 does not contributing to connection-level flow control (#204)
- o Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- o Remove connection-level flow control exclusion for some streams (except 1) (#246)
- o RST_STREAM affects connection-level flow control (#162, #163)
- o Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- o Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- o Added the ability to pad between frames (#158, #276)
- o Remove error code and reason phrase from GOAWAY (#352, #355)
- o GOAWAY includes a final stream number for both directions (#347)
- o Error codes for RST_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)
- o Defined priority as the responsibility of the application protocol (#104, #303)

C.2. Since draft-ietf-quic-transport-00:

- o Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- o Defined versioning
- o Reworked description of packet and frame layout
- o Error code space is divided into regions for each component
- o Use big endian for all numeric values

C.3. Since draft-hamilton-quic-transport-protocol-01:

- o Adopted as base for draft-ietf-quic-tls.
- o Updated authors/editors list.
- o Added IANA Considerations section.
- o Moved Contributors and Acknowledgments to appendices.

Authors' Addresses

Jana Iyengar (editor)
Google

Email: jri@google.com

Martin Thomson (editor)
Mozilla

Email: martin.thomson@gmail.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 19 July 2021

J. Iyengar, Ed.
Fastly
M. Thomson, Ed.
Mozilla
15 January 2021

QUIC: A UDP-Based Multiplexed and Secure Transport
draft-ietf-quic-transport-34

Abstract

This document defines the core of the QUIC transport protocol. QUIC provides applications with flow-controlled streams for structured communication, low-latency connection establishment, and network path migration. QUIC includes security measures that ensure confidentiality, integrity, and availability in a range of deployment circumstances. Accompanying documents describe the integration of TLS for key negotiation, loss detection, and an exemplary congestion control algorithm.

DO NOT DEPLOY THIS VERSION OF QUIC

DO NOT DEPLOY THIS VERSION OF QUIC UNTIL IT IS IN AN RFC. This version is still a work in progress. For trial deployments, please use earlier versions.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org (<mailto:quic@ietf.org>)), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-transport>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 July 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Overview	7
1.1. Document Structure	8
1.2. Terms and Definitions	10
1.3. Notational Conventions	11
2. Streams	12
2.1. Stream Types and Identifiers	13
2.2. Sending and Receiving Data	14
2.3. Stream Prioritization	14
2.4. Operations on Streams	15
3. Stream States	15
3.1. Sending Stream States	16
3.2. Receiving Stream States	18
3.3. Permitted Frame Types	21
3.4. Bidirectional Stream States	21
3.5. Solicited State Transitions	23
4. Flow Control	24
4.1. Data Flow Control	24
4.2. Increasing Flow Control Limits	25
4.3. Flow Control Performance	26
4.4. Handling Stream Cancellation	27
4.5. Stream Final Size	27
4.6. Controlling Concurrency	28
5. Connections	29
5.1. Connection ID	29
5.1.1. Issuing Connection IDs	31

5.1.2.	Consuming and Retiring Connection IDs	32
5.2.	Matching Packets to Connections	33
5.2.1.	Client Packet Handling	34
5.2.2.	Server Packet Handling	35
5.2.3.	Considerations for Simple Load Balancers	35
5.3.	Operations on Connections	36
6.	Version Negotiation	37
6.1.	Sending Version Negotiation Packets	37
6.2.	Handling Version Negotiation Packets	38
6.2.1.	Version Negotiation Between Draft Versions	38
6.3.	Using Reserved Versions	39
7.	Cryptographic and Transport Handshake	39
7.1.	Example Handshake Flows	41
7.2.	Negotiating Connection IDs	42
7.3.	Authenticating Connection IDs	43
7.4.	Transport Parameters	45
7.4.1.	Values of Transport Parameters for 0-RTT	46
7.4.2.	New Transport Parameters	48
7.5.	Cryptographic Message Buffering	49
8.	Address Validation	49
8.1.	Address Validation During Connection Establishment	50
8.1.1.	Token Construction	51
8.1.2.	Address Validation using Retry Packets	51
8.1.3.	Address Validation for Future Connections	52
8.1.4.	Address Validation Token Integrity	55
8.2.	Path Validation	55
8.2.1.	Initiating Path Validation	56
8.2.2.	Path Validation Responses	57
8.2.3.	Successful Path Validation	58
8.2.4.	Failed Path Validation	58
9.	Connection Migration	59
9.1.	Probing a New Path	60
9.2.	Initiating Connection Migration	60
9.3.	Responding to Connection Migration	61
9.3.1.	Peer Address Spoofing	62
9.3.2.	On-Path Address Spoofing	62
9.3.3.	Off-Path Packet Forwarding	63
9.4.	Loss Detection and Congestion Control	64
9.5.	Privacy Implications of Connection Migration	65
9.6.	Server's Preferred Address	66
9.6.1.	Communicating a Preferred Address	66
9.6.2.	Migration to a Preferred Address	67
9.6.3.	Interaction of Client Migration and Preferred Address	67
9.7.	Use of IPv6 Flow-Label and Migration	68
10.	Connection Termination	69
10.1.	Idle Timeout	69
10.1.1.	Liveness Testing	69

10.1.2.	Deferring Idle Timeout	70
10.2.	Immediate Close	70
10.2.1.	Closing Connection State	71
10.2.2.	Draining Connection State	72
10.2.3.	Immediate Close During the Handshake	73
10.3.	Stateless Reset	74
10.3.1.	Detecting a Stateless Reset	77
10.3.2.	Calculating a Stateless Reset Token	78
10.3.3.	Looping	79
11.	Error Handling	79
11.1.	Connection Errors	80
11.2.	Stream Errors	81
12.	Packets and Frames	81
12.1.	Protected Packets	82
12.2.	Coalescing Packets	82
12.3.	Packet Numbers	83
12.4.	Frames and Frame Types	85
12.5.	Frames and Number Spaces	89
13.	Packetization and Reliability	90
13.1.	Packet Processing	90
13.2.	Generating Acknowledgments	91
13.2.1.	Sending ACK Frames	91
13.2.2.	Acknowledgment Frequency	92
13.2.3.	Managing ACK Ranges	93
13.2.4.	Limiting Ranges by Tracking ACK Frames	94
13.2.5.	Measuring and Reporting Host Delay	95
13.2.6.	ACK Frames and Packet Protection	95
13.2.7.	PADDING Frames Consume Congestion Window	95
13.3.	Retransmission of Information	96
13.4.	Explicit Congestion Notification	98
13.4.1.	Reporting ECN Counts	99
13.4.2.	ECN Validation	99
14.	Datagram Size	101
14.1.	Initial Datagram Size	102
14.2.	Path Maximum Transmission Unit	103
14.2.1.	Handling of ICMP Messages by PMTUD	104
14.3.	Datagram Packetization Layer PMTU Discovery	105
14.3.1.	DPLPMTUD and Initial Connectivity	105
14.3.2.	Validating the Network Path with DPLPMTUD	105
14.3.3.	Handling of ICMP Messages by DPLPMTUD	105
14.4.	Sending QUIC PMTU Probes	105
14.4.1.	PMTU Probes Containing Source Connection ID	106
15.	Versions	106
16.	Variable-Length Integer Encoding	107
17.	Packet Formats	108
17.1.	Packet Number Encoding and Decoding	108
17.2.	Long Header Packets	109
17.2.1.	Version Negotiation Packet	112

17.2.2.	Initial Packet	113
17.2.3.	0-RTT	115
17.2.4.	Handshake Packet	117
17.2.5.	Retry Packet	118
17.3.	Short Header Packets	120
17.3.1.	1-RTT Packet	120
17.4.	Latency Spin Bit	122
18.	Transport Parameter Encoding	123
18.1.	Reserved Transport Parameters	124
18.2.	Transport Parameter Definitions	124
19.	Frame Types and Formats	128
19.1.	PADDING Frames	129
19.2.	PING Frames	129
19.3.	ACK Frames	130
19.3.1.	ACK Ranges	131
19.3.2.	ECN Counts	132
19.4.	RESET_STREAM Frames	133
19.5.	STOP_SENDING Frames	134
19.6.	CRYPTO Frames	135
19.7.	NEW_TOKEN Frames	136
19.8.	STREAM Frames	136
19.9.	MAX_DATA Frames	138
19.10.	MAX_STREAM_DATA Frames	138
19.11.	MAX_STREAMS Frames	139
19.12.	DATA_BLOCKED Frames	140
19.13.	STREAM_DATA_BLOCKED Frames	141
19.14.	STREAMS_BLOCKED Frames	141
19.15.	NEW_CONNECTION_ID Frames	142
19.16.	RETIRE_CONNECTION_ID Frames	144
19.17.	PATH_CHALLENGE Frames	145
19.18.	PATH_RESPONSE Frames	145
19.19.	CONNECTION_CLOSE Frames	146
19.20.	HANDSHAKE_DONE Frames	147
19.21.	Extension Frames	147
20.	Error Codes	148
20.1.	Transport Error Codes	148
20.2.	Application Protocol Error Codes	150
21.	Security Considerations	150
21.1.	Overview of Security Properties	150
21.1.1.	Handshake	151
21.1.2.	Protected Packets	153
21.1.3.	Connection Migration	153
21.2.	Handshake Denial of Service	158
21.3.	Amplification Attack	159
21.4.	Optimistic ACK Attack	159
21.5.	Request Forgery Attacks	159
21.5.1.	Control Options for Endpoints	160
21.5.2.	Request Forgery with Client Initial Packets	161

21.5.3.	Request Forgery with Preferred Addresses	162
21.5.4.	Request Forgery with Spoofed Migration	162
21.5.5.	Request Forgery with Version Negotiation	163
21.5.6.	Generic Request Forgery Countermeasures	163
21.6.	Slowloris Attacks	164
21.7.	Stream Fragmentation and Reassembly Attacks	165
21.8.	Stream Commitment Attack	165
21.9.	Peer Denial of Service	166
21.10.	Explicit Congestion Notification Attacks	166
21.11.	Stateless Reset Oracle	167
21.12.	Version Downgrade	167
21.13.	Targeted Attacks by Routing	168
21.14.	Traffic Analysis	168
22.	IANA Considerations	168
22.1.	Registration Policies for QUIC Registries	168
22.1.1.	Provisional Registrations	168
22.1.2.	Selecting Codepoints	169
22.1.3.	Reclaiming Provisional Codepoints	170
22.1.4.	Permanent Registrations	170
22.2.	QUIC Versions Registry	171
22.3.	QUIC Transport Parameter Registry	172
22.4.	QUIC Frame Types Registry	173
22.5.	QUIC Transport Error Codes Registry	174
23.	References	176
23.1.	Normative References	176
23.2.	Informative References	178
Appendix A.	Pseudocode	181
A.1.	Sample Variable-Length Integer Decoding	181
A.2.	Sample Packet Number Encoding Algorithm	182
A.3.	Sample Packet Number Decoding Algorithm	182
A.4.	Sample ECN Validation Algorithm	183
Appendix B.	Change Log	184
B.1.	Since draft-ietf-quic-transport-32	184
B.2.	Since draft-ietf-quic-transport-31	185
B.3.	Since draft-ietf-quic-transport-30	185
B.4.	Since draft-ietf-quic-transport-29	186
B.5.	Since draft-ietf-quic-transport-28	186
B.6.	Since draft-ietf-quic-transport-27	187
B.7.	Since draft-ietf-quic-transport-26	188
B.8.	Since draft-ietf-quic-transport-25	188
B.9.	Since draft-ietf-quic-transport-24	188
B.10.	Since draft-ietf-quic-transport-23	189
B.11.	Since draft-ietf-quic-transport-22	190
B.12.	Since draft-ietf-quic-transport-21	191
B.13.	Since draft-ietf-quic-transport-20	191
B.14.	Since draft-ietf-quic-transport-19	192
B.15.	Since draft-ietf-quic-transport-18	192
B.16.	Since draft-ietf-quic-transport-17	193

B.17. Since draft-ietf-quic-transport-16	194
B.18. Since draft-ietf-quic-transport-15	195
B.19. Since draft-ietf-quic-transport-14	195
B.20. Since draft-ietf-quic-transport-13	195
B.21. Since draft-ietf-quic-transport-12	196
B.22. Since draft-ietf-quic-transport-11	197
B.23. Since draft-ietf-quic-transport-10	197
B.24. Since draft-ietf-quic-transport-09	198
B.25. Since draft-ietf-quic-transport-08	199
B.26. Since draft-ietf-quic-transport-07	199
B.27. Since draft-ietf-quic-transport-06	200
B.28. Since draft-ietf-quic-transport-05	201
B.29. Since draft-ietf-quic-transport-04	201
B.30. Since draft-ietf-quic-transport-03	202
B.31. Since draft-ietf-quic-transport-02	202
B.32. Since draft-ietf-quic-transport-01	203
B.33. Since draft-ietf-quic-transport-00	205
B.34. Since draft-hamilton-quic-transport-protocol-01	205
Contributors	205
Authors' Addresses	207

1. Overview

QUIC is a secure general-purpose transport protocol. This document defines version 1 of QUIC, which conforms to the version-independent properties of QUIC defined in [QUIC-INVARIANTS].

QUIC is a connection-oriented protocol that creates a stateful interaction between a client and server.

The QUIC handshake combines negotiation of cryptographic and transport parameters. QUIC integrates the TLS ([TLS13]) handshake, although using a customized framing for protecting packets. The integration of TLS and QUIC is described in more detail in [QUIC-TLS]. The handshake is structured to permit the exchange of application data as soon as possible. This includes an option for clients to send data immediately (0-RTT), which requires some form of prior communication or configuration to enable.

Endpoints communicate in QUIC by exchanging QUIC packets. Most packets contain frames, which carry control information and application data between endpoints. QUIC authenticates the entirety of each packet and encrypts as much of each packet as is practical. QUIC packets are carried in UDP datagrams ([UDP]) to better facilitate deployment in existing systems and networks.

Application protocols exchange information over a QUIC connection via streams, which are ordered sequences of bytes. Two types of stream can be created: bidirectional streams, which allow both endpoints to send data; and unidirectional streams, which allow a single endpoint to send data. A credit-based scheme is used to limit stream creation and to bound the amount of data that can be sent.

QUIC provides the necessary feedback to implement reliable delivery and congestion control. An algorithm for detecting and recovering from loss of data is described in [QUIC-RECOVERY]. QUIC depends on congestion control to avoid network congestion. An exemplary congestion control algorithm is also described in [QUIC-RECOVERY].

QUIC connections are not strictly bound to a single network path. Connection migration uses connection identifiers to allow connections to transfer to a new network path. Only clients are able to migrate in this version of QUIC. This design also allows connections to continue after changes in network topology or address mappings, such as might be caused by NAT rebinding.

Once established, multiple options are provided for connection termination. Applications can manage a graceful shutdown, endpoints can negotiate a timeout period, errors can cause immediate connection teardown, and a stateless mechanism provides for termination of connections after one endpoint has lost state.

1.1. Document Structure

This document describes the core QUIC protocol and is structured as follows:

- * Streams are the basic service abstraction that QUIC provides.
 - Section 2 describes core concepts related to streams,
 - Section 3 provides a reference model for stream states, and
 - Section 4 outlines the operation of flow control.
- * Connections are the context in which QUIC endpoints communicate.
 - Section 5 describes core concepts related to connections,
 - Section 6 describes version negotiation,
 - Section 7 details the process for establishing connections,

- Section 8 describes address validation and critical denial of service mitigations,
 - Section 9 describes how endpoints migrate a connection to a new network path,
 - Section 10 lists the options for terminating an open connection, and
 - Section 11 provides guidance for stream and connection error handling.
- * Packets and frames are the basic unit used by QUIC to communicate.
- Section 12 describes concepts related to packets and frames,
 - Section 13 defines models for the transmission, retransmission, and acknowledgment of data, and
 - Section 14 specifies rules for managing the size of datagrams carrying QUIC packets.
- * Finally, encoding details of QUIC protocol elements are described in:
- Section 15 (Versions),
 - Section 16 (Integer Encoding),
 - Section 17 (Packet Headers),
 - Section 18 (Transport Parameters),
 - Section 19 (Frames), and
 - Section 20 (Errors).

Accompanying documents describe QUIC's loss detection and congestion control [QUIC-RECOVERY], and the use of TLS and other cryptographic mechanisms [QUIC-TLS].

This document defines QUIC version 1, which conforms to the protocol invariants in [QUIC-INVARIANTS].

To refer to QUIC version 1, cite this document. References to the limited set of version-independent properties of QUIC can cite [QUIC-INVARIANTS].

1.2. Terms and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Commonly used terms in the document are described below.

QUIC: The transport protocol described by this document. QUIC is a name, not an acronym.

Endpoint: An entity that can participate in a QUIC connection by generating, receiving, and processing QUIC packets. There are only two types of endpoint in QUIC: client and server.

Client: The endpoint that initiates a QUIC connection.

Server: The endpoint that accepts a QUIC connection.

QUIC packet: A complete processable unit of QUIC that can be encapsulated in a UDP datagram. One or more QUIC packets can be encapsulated in a single UDP datagram.

Ack-eliciting Packet: A QUIC packet that contains frames other than ACK, PADDING, and CONNECTION_CLOSE. These cause a recipient to send an acknowledgment; see Section 13.2.1.

Frame: A unit of structured protocol information. There are multiple frame types, each of which carries different information. Frames are contained in QUIC packets.

Address: When used without qualification, the tuple of IP version, IP address, and UDP port number that represents one end of a network path.

Connection ID: An identifier that is used to identify a QUIC connection at an endpoint. Each endpoint selects one or more Connection IDs for its peer to include in packets sent towards the endpoint. This value is opaque to the peer.

Stream: A unidirectional or bidirectional channel of ordered bytes within a QUIC connection. A QUIC connection can carry multiple simultaneous streams.

Application: An entity that uses QUIC to send and receive data.

This document uses the terms "QUIC packets", "UDP datagrams", and "IP packets" to refer to the units of the respective protocols. That is, one or more QUIC packets can be encapsulated in a UDP datagram, which is in turn encapsulated in an IP packet.

1.3. Notational Conventions

Packet and frame diagrams in this document use a custom format. The purpose of this format is to summarize, not define, protocol elements. Prose defines the complete semantics and details of structures.

Complex fields are named and then followed by a list of fields surrounded by a pair of matching braces. Each field in this list is separated by commas.

Individual fields include length information, plus indications about fixed value, optionality, or repetitions. Individual fields use the following notational conventions, with all lengths in bits:

- x (A): Indicates that x is A bits long
- x (i): Indicates that x holds an integer value using the variable-length encoding in Section 16
- x (A..B): Indicates that x can be any length from A to B; A can be omitted to indicate a minimum of zero bits and B can be omitted to indicate no set upper limit; values in this format always end on an byte boundary
- x (L) = C: Indicates that x has a fixed value of C with the length described by L, which can use any of the three length forms above
- x (L) = C..D: Indicates that x has a value in the range from C to D, inclusive, with the length described by L, as above
- [x (L)]: Indicates that x is optional (and has length of L)
- x (L) ...: Indicates that zero or more instances of x are present (and that each instance is length L)

This document uses network byte order (that is, big endian) values. Fields are placed starting from the high-order bits of each byte.

By convention, individual fields reference a complex field by using the name of the complex field.

For example:

```
Example Structure {  
  One-bit Field (1),  
  7-bit Field with Fixed Value (7) = 61,  
  Field with Variable-Length Integer (i),  
  Arbitrary-Length Field (...),  
  Variable-Length Field (8..24),  
  Field With Minimum Length (16...),  
  Field With Maximum Length (...128),  
  [Optional Field (64)],  
  Repeated Field (8) ...,  
}
```

Figure 1: Example Format

When a single-bit field is referenced in prose, the position of that field can be clarified by using the value of the byte that carries the field with the field's value set. For example, the value 0x80 could be used to refer to the single-bit field in the most significant bit of the byte, such as One-bit Field in Figure 1.

2. Streams

Streams in QUIC provide a lightweight, ordered byte-stream abstraction to an application. Streams can be unidirectional or bidirectional.

Streams can be created by sending data. Other processes associated with stream management - ending, cancelling, and managing flow control - are all designed to impose minimal overheads. For instance, a single STREAM frame (Section 19.8) can open, carry data for, and close a stream. Streams can also be long-lived and can last the entire duration of a connection.

Streams can be created by either endpoint, can concurrently send data interleaved with other streams, and can be cancelled. QUIC does not provide any means of ensuring ordering between bytes on different streams.

QUIC allows for an arbitrary number of streams to operate concurrently and for an arbitrary amount of data to be sent on any stream, subject to flow control constraints and stream limits; see Section 4.

2.1. Stream Types and Identifiers

Streams can be unidirectional or bidirectional. Unidirectional streams carry data in one direction: from the initiator of the stream to its peer. Bidirectional streams allow for data to be sent in both directions.

Streams are identified within a connection by a numeric value, referred to as the stream ID. A stream ID is a 62-bit integer (0 to $2^{62}-1$) that is unique for all streams on a connection. Stream IDs are encoded as variable-length integers; see Section 16. A QUIC endpoint MUST NOT reuse a stream ID within a connection.

The least significant bit (0x1) of the stream ID identifies the initiator of the stream. Client-initiated streams have even-numbered stream IDs (with the bit set to 0), and server-initiated streams have odd-numbered stream IDs (with the bit set to 1).

The second least significant bit (0x2) of the stream ID distinguishes between bidirectional streams (with the bit set to 0) and unidirectional streams (with the bit set to 1).

The two least significant bits from a stream ID therefore identify a stream as one of four types, as summarized in Table 1.

Bits	Stream Type
0x0	Client-Initiated, Bidirectional
0x1	Server-Initiated, Bidirectional
0x2	Client-Initiated, Unidirectional
0x3	Server-Initiated, Unidirectional

Table 1: Stream ID Types

The stream space for each type begins at the minimum value (0x0 through 0x3 respectively); successive streams of each type are created with numerically increasing stream IDs. A stream ID that is used out of order results in all streams of that type with lower-numbered stream IDs also being opened.

2.2. Sending and Receiving Data

STREAM frames (Section 19.8) encapsulate data sent by an application. An endpoint uses the Stream ID and Offset fields in STREAM frames to place data in order.

Endpoints **MUST** be able to deliver stream data to an application as an ordered byte-stream. Delivering an ordered byte-stream requires that an endpoint buffer any data that is received out of order, up to the advertised flow control limit.

QUIC makes no specific allowances for delivery of stream data out of order. However, implementations **MAY** choose to offer the ability to deliver data out of order to a receiving application.

An endpoint could receive data for a stream at the same stream offset multiple times. Data that has already been received can be discarded. The data at a given offset **MUST NOT** change if it is sent multiple times; an endpoint **MAY** treat receipt of different data at the same offset within a stream as a connection error of type `PROTOCOL_VIOLATION`.

Streams are an ordered byte-stream abstraction with no other structure visible to QUIC. STREAM frame boundaries are not expected to be preserved when data is transmitted, retransmitted after packet loss, or delivered to the application at a receiver.

An endpoint **MUST NOT** send data on any stream without ensuring that it is within the flow control limits set by its peer. Flow control is described in detail in Section 4.

2.3. Stream Prioritization

Stream multiplexing can have a significant effect on application performance if resources allocated to streams are correctly prioritized.

QUIC does not provide a mechanism for exchanging prioritization information. Instead, it relies on receiving priority information from the application.

A QUIC implementation **SHOULD** provide ways in which an application can indicate the relative priority of streams. An implementation uses information provided by the application to determine how to allocate resources to active streams.

2.4. Operations on Streams

This document does not define an API for QUIC, but instead defines a set of functions on streams that application protocols can rely upon. An application protocol can assume that a QUIC implementation provides an interface that includes the operations described in this section. An implementation designed for use with a specific application protocol might provide only those operations that are used by that protocol.

On the sending part of a stream, an application protocol can:

- * write data, understanding when stream flow control credit (Section 4.1) has successfully been reserved to send the written data;
- * end the stream (clean termination), resulting in a STREAM frame (Section 19.8) with the FIN bit set; and
- * reset the stream (abrupt termination), resulting in a RESET_STREAM frame (Section 19.4) if the stream was not already in a terminal state.

On the receiving part of a stream, an application protocol can:

- * read data; and
- * abort reading of the stream and request closure, possibly resulting in a STOP_SENDING frame (Section 19.5).

An application protocol can also request to be informed of state changes on streams, including when the peer has opened or reset a stream, when a peer aborts reading on a stream, when new data is available, and when data can or cannot be written to the stream due to flow control.

3. Stream States

This section describes streams in terms of their send or receive components. Two state machines are described: one for the streams on which an endpoint transmits data (Section 3.1), and another for streams on which an endpoint receives data (Section 3.2).

Unidirectional streams use either the sending or receiving state machine depending on the stream type and endpoint role. Bidirectional streams use both state machines at both endpoints. For the most part, the use of these state machines is the same whether the stream is unidirectional or bidirectional. The conditions for

opening a stream are slightly more complex for a bidirectional stream because the opening of either the send or receive side causes the stream to open in both directions.

The state machines shown in this section are largely informative. This document uses stream states to describe rules for when and how different types of frames can be sent and the reactions that are expected when different types of frames are received. Though these state machines are intended to be useful in implementing QUIC, these states are not intended to constrain implementations. An implementation can define a different state machine as long as its behavior is consistent with an implementation that implements these states.

Note: In some cases, a single event or action can cause a transition through multiple states. For instance, sending STREAM with a FIN bit set can cause two state transitions for a sending stream: from the Ready state to the Send state, and from the Send state to the Data Sent state.

3.1. Sending Stream States

Figure 2 shows the states for the part of a stream that sends data to a peer.

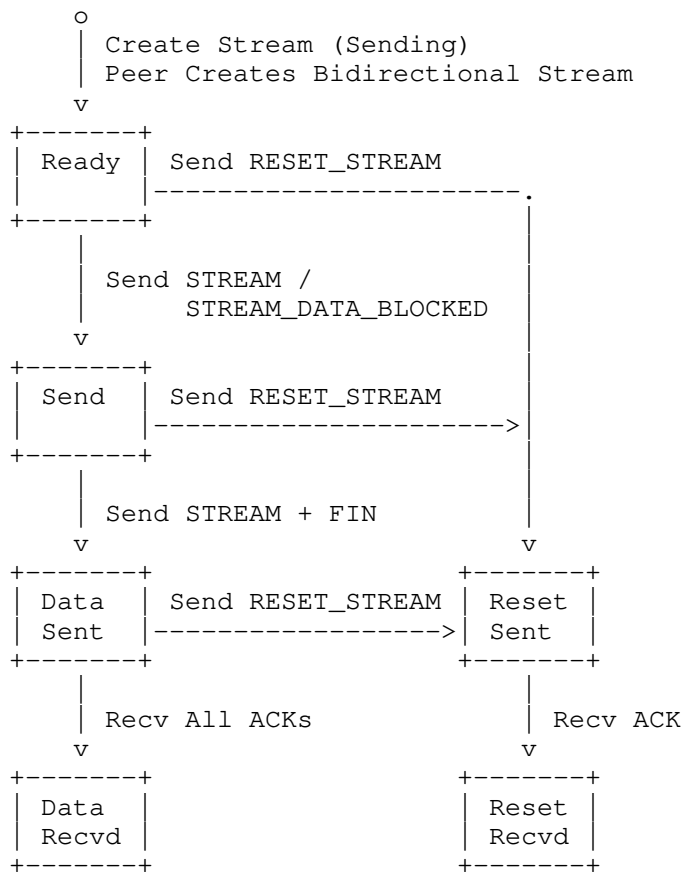


Figure 2: States for Sending Parts of Streams

The sending part of a stream that the endpoint initiates (types 0 and 2 for clients, 1 and 3 for servers) is opened by the application. The "Ready" state represents a newly created stream that is able to accept data from the application. Stream data might be buffered in this state in preparation for sending.

Sending the first STREAM or STREAM_DATA_BLOCKED frame causes a sending part of a stream to enter the "Send" state. An implementation might choose to defer allocating a stream ID to a stream until it sends the first STREAM frame and enters this state, which can allow for better stream prioritization.

The sending part of a bidirectional stream initiated by a peer (type 0 for a server, type 1 for a client) starts in the "Ready" state when the receiving part is created.

In the "Send" state, an endpoint transmits - and retransmits as necessary - stream data in STREAM frames. The endpoint respects the flow control limits set by its peer, and continues to accept and process MAX_STREAM_DATA frames. An endpoint in the "Send" state generates STREAM_DATA_BLOCKED frames if it is blocked from sending by stream flow control limits (Section 4.1).

After the application indicates that all stream data has been sent and a STREAM frame containing the FIN bit is sent, the sending part of the stream enters the "Data Sent" state. From this state, the endpoint only retransmits stream data as necessary. The endpoint does not need to check flow control limits or send STREAM_DATA_BLOCKED frames for a stream in this state. MAX_STREAM_DATA frames might be received until the peer receives the final stream offset. The endpoint can safely ignore any MAX_STREAM_DATA frames it receives from its peer for a stream in this state.

Once all stream data has been successfully acknowledged, the sending part of the stream enters the "Data Recvd" state, which is a terminal state.

From any of the "Ready", "Send", or "Data Sent" states, an application can signal that it wishes to abandon transmission of stream data. Alternatively, an endpoint might receive a STOP_SENDING frame from its peer. In either case, the endpoint sends a RESET_STREAM frame, which causes the stream to enter the "Reset Sent" state.

An endpoint MAY send a RESET_STREAM as the first frame that mentions a stream; this causes the sending part of that stream to open and then immediately transition to the "Reset Sent" state.

Once a packet containing a RESET_STREAM has been acknowledged, the sending part of the stream enters the "Reset Recvd" state, which is a terminal state.

3.2. Receiving Stream States

Figure 3 shows the states for the part of a stream that receives data from a peer. The states for a receiving part of a stream mirror only some of the states of the sending part of the stream at the peer. The receiving part of a stream does not track states on the sending part that cannot be observed, such as the "Ready" state. Instead, the receiving part of a stream tracks the delivery of data to the application, some of which cannot be observed by the sender.

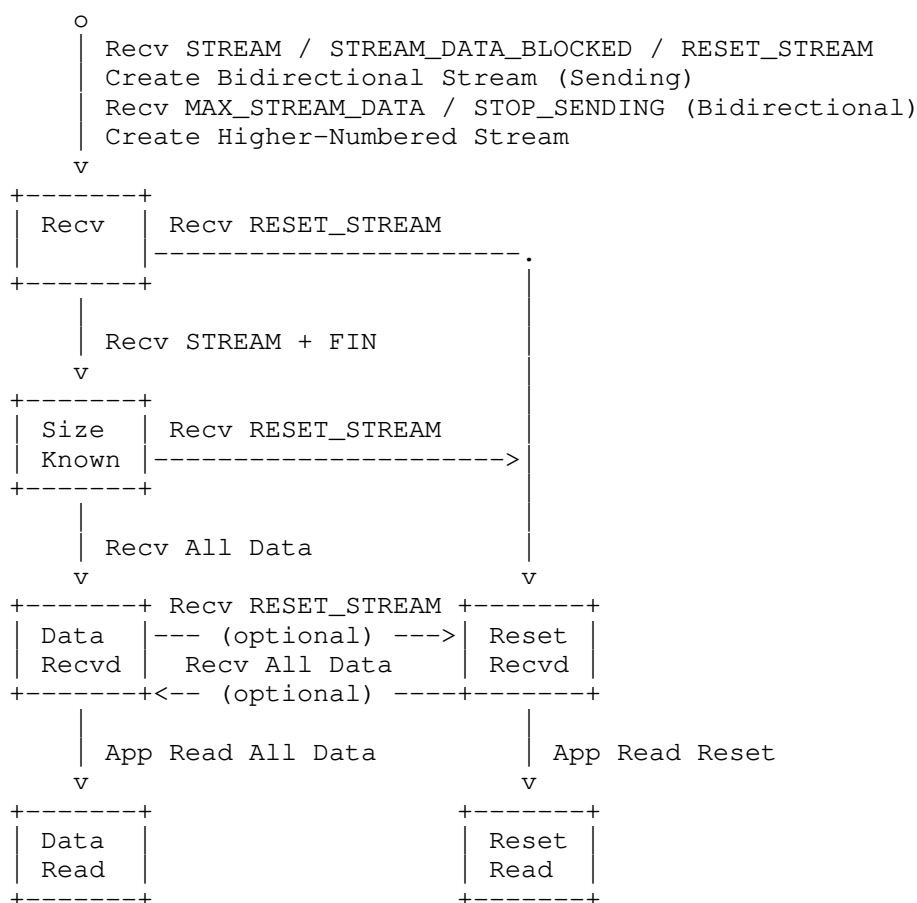


Figure 3: States for Receiving Parts of Streams

The receiving part of a stream initiated by a peer (types 1 and 3 for a client, or 0 and 2 for a server) is created when the first STREAM, STREAM_DATA_BLOCKED, or RESET_STREAM frame is received for that stream. For bidirectional streams initiated by a peer, receipt of a MAX_STREAM_DATA or STOP_SENDING frame for the sending part of the stream also creates the receiving part. The initial state for the receiving part of a stream is "Recv".

For a bidirectional stream, the receiving part enters the "Recv" state when the sending part initiated by the endpoint (type 0 for a client, type 1 for a server) enters the "Ready" state.

An endpoint opens a bidirectional stream when a `MAX_STREAM_DATA` or `STOP_SENDING` frame is received from the peer for that stream. Receiving a `MAX_STREAM_DATA` frame for an unopened stream indicates that the remote peer has opened the stream and is providing flow control credit. Receiving a `STOP_SENDING` frame for an unopened stream indicates that the remote peer no longer wishes to receive data on this stream. Either frame might arrive before a `STREAM` or `STREAM_DATA_BLOCKED` frame if packets are lost or reordered.

Before a stream is created, all streams of the same type with lower-numbered stream IDs **MUST** be created. This ensures that the creation order for streams is consistent on both endpoints.

In the "Recv" state, the endpoint receives `STREAM` and `STREAM_DATA_BLOCKED` frames. Incoming data is buffered and can be reassembled into the correct order for delivery to the application. As data is consumed by the application and buffer space becomes available, the endpoint sends `MAX_STREAM_DATA` frames to allow the peer to send more data.

When a `STREAM` frame with a FIN bit is received, the final size of the stream is known; see Section 4.5. The receiving part of the stream then enters the "Size Known" state. In this state, the endpoint no longer needs to send `MAX_STREAM_DATA` frames, it only receives any retransmissions of stream data.

Once all data for the stream has been received, the receiving part enters the "Data Recvd" state. This might happen as a result of receiving the same `STREAM` frame that causes the transition to "Size Known". After all data has been received, any `STREAM` or `STREAM_DATA_BLOCKED` frames for the stream can be discarded.

The "Data Recvd" state persists until stream data has been delivered to the application. Once stream data has been delivered, the stream enters the "Data Read" state, which is a terminal state.

Receiving a `RESET_STREAM` frame in the "Recv" or "Size Known" states causes the stream to enter the "Reset Recvd" state. This might cause the delivery of stream data to the application to be interrupted.

It is possible that all stream data has already been received when a `RESET_STREAM` is received (that is, in the "Data Recvd" state). Similarly, it is possible for remaining stream data to arrive after receiving a `RESET_STREAM` frame (the "Reset Recvd" state). An implementation is free to manage this situation as it chooses.

Sending RESET_STREAM means that an endpoint cannot guarantee delivery of stream data; however there is no requirement that stream data not be delivered if a RESET_STREAM is received. An implementation MAY interrupt delivery of stream data, discard any data that was not consumed, and signal the receipt of the RESET_STREAM. A RESET_STREAM signal might be suppressed or withheld if stream data is completely received and is buffered to be read by the application. If the RESET_STREAM is suppressed, the receiving part of the stream remains in "Data Recvd".

Once the application receives the signal indicating that the stream was reset, the receiving part of the stream transitions to the "Reset Read" state, which is a terminal state.

3.3. Permitted Frame Types

The sender of a stream sends just three frame types that affect the state of a stream at either sender or receiver: STREAM (Section 19.8), STREAM_DATA_BLOCKED (Section 19.13), and RESET_STREAM (Section 19.4).

A sender MUST NOT send any of these frames from a terminal state ("Data Recvd" or "Reset Recvd"). A sender MUST NOT send a STREAM or STREAM_DATA_BLOCKED frame for a stream in the "Reset Sent" state or any terminal state, that is, after sending a RESET_STREAM frame. A receiver could receive any of these three frames in any state, due to the possibility of delayed delivery of packets carrying them.

The receiver of a stream sends MAX_STREAM_DATA (Section 19.10) and STOP_SENDING frames (Section 19.5).

The receiver only sends MAX_STREAM_DATA in the "Recv" state. A receiver MAY send STOP_SENDING in any state where it has not received a RESET_STREAM frame; that is states other than "Reset Recvd" or "Reset Read". However there is little value in sending a STOP_SENDING frame in the "Data Recvd" state, since all stream data has been received. A sender could receive either of these two frames in any state as a result of delayed delivery of packets.

3.4. Bidirectional Stream States

A bidirectional stream is composed of sending and receiving parts. Implementations can represent states of the bidirectional stream as composites of sending and receiving stream states. The simplest model presents the stream as "open" when either sending or receiving parts are in a non-terminal state and "closed" when both sending and receiving streams are in terminal states.

Table 2 shows a more complex mapping of bidirectional stream states that loosely correspond to the stream states in HTTP/2 [HTTP2]. This shows that multiple states on sending or receiving parts of streams are mapped to the same composite state. Note that this is just one possibility for such a mapping; this mapping requires that data is acknowledged before the transition to a "closed" or "half-closed" state.

Sending Part	Receiving Part	Composite State
No Stream/Ready	No Stream/Recv *1	idle
Ready/Send/Data Sent	Recv/Size Known	open
Ready/Send/Data Sent	Data Recvd/Data Read	half-closed (remote)
Ready/Send/Data Sent	Reset Recvd/Reset Read	half-closed (remote)
Data Recvd	Recv/Size Known	half-closed (local)
Reset Sent/Reset Recvd	Recv/Size Known	half-closed (local)
Reset Sent/Reset Recvd	Data Recvd/Data Read	closed
Reset Sent/Reset Recvd	Reset Recvd/Reset Read	closed
Data Recvd	Data Recvd/Data Read	closed
Data Recvd	Reset Recvd/Reset Read	closed

Table 2: Possible Mapping of Stream States to HTTP/2

Note (*1): A stream is considered "idle" if it has not yet been created, or if the receiving part of the stream is in the "Recv" state without yet having received any frames.

3.5. Solicited State Transitions

If an application is no longer interested in the data it is receiving on a stream, it can abort reading the stream and specify an application error code.

If the stream is in the "Recv" or "Size Known" states, the transport SHOULD signal this by sending a STOP_SENDING frame to prompt closure of the stream in the opposite direction. This typically indicates that the receiving application is no longer reading data it receives from the stream, but it is not a guarantee that incoming data will be ignored.

STREAM frames received after sending a STOP_SENDING frame are still counted toward connection and stream flow control, even though these frames can be discarded upon receipt.

A STOP_SENDING frame requests that the receiving endpoint send a RESET_STREAM frame. An endpoint that receives a STOP_SENDING frame MUST send a RESET_STREAM frame if the stream is in the Ready or Send state. If the stream is in the "Data Sent" state, the endpoint MAY defer sending the RESET_STREAM frame until the packets containing outstanding data are acknowledged or declared lost. If any outstanding data is declared lost, the endpoint SHOULD send a RESET_STREAM frame instead of retransmitting the data.

An endpoint SHOULD copy the error code from the STOP_SENDING frame to the RESET_STREAM frame it sends, but can use any application error code. An endpoint that sends a STOP_SENDING frame MAY ignore the error code in any RESET_STREAM frames subsequently received for that stream.

STOP_SENDING SHOULD only be sent for a stream that has not been reset by the peer. STOP_SENDING is most useful for streams in the "Recv" or "Size Known" states.

An endpoint is expected to send another STOP_SENDING frame if a packet containing a previous STOP_SENDING is lost. However, once either all stream data or a RESET_STREAM frame has been received for the stream - that is, the stream is in any state other than "Recv" or "Size Known" - sending a STOP_SENDING frame is unnecessary.

An endpoint that wishes to terminate both directions of a bidirectional stream can terminate one direction by sending a RESET_STREAM frame, and it can encourage prompt termination in the opposite direction by sending a STOP_SENDING frame.

4. Flow Control

Receivers need to limit the amount of data that they are required to buffer, in order to prevent a fast sender from overwhelming them or a malicious sender from consuming a large amount of memory. To enable a receiver to limit memory commitments for a connection, streams are flow controlled both individually and across a connection as a whole. A QUIC receiver controls the maximum amount of data the sender can send on a stream as well as across all streams at any time, as described in Section 4.1 and Section 4.2.

Similarly, to limit concurrency within a connection, a QUIC endpoint controls the maximum cumulative number of streams that its peer can initiate, as described in Section 4.6.

Data sent in CRYPTO frames is not flow controlled in the same way as stream data. QUIC relies on the cryptographic protocol implementation to avoid excessive buffering of data; see [QUIC-TLS]. To avoid excessive buffering at multiple layers, QUIC implementations SHOULD provide an interface for the cryptographic protocol implementation to communicate its buffering limits.

4.1. Data Flow Control

QUIC employs a limit-based flow-control scheme where a receiver advertises the limit of total bytes it is prepared to receive on a given stream or for the entire connection. This leads to two levels of data flow control in QUIC:

- * Stream flow control, which prevents a single stream from consuming the entire receive buffer for a connection by limiting the amount of data that can be sent on each stream.
- * Connection flow control, which prevents senders from exceeding a receiver's buffer capacity for the connection, by limiting the total bytes of stream data sent in STREAM frames on all streams.

Senders MUST NOT send data in excess of either limit.

A receiver sets initial limits for all streams through transport parameters during the handshake (Section 7.4). Subsequently, a receiver sends MAX_STREAM_DATA (Section 19.10) or MAX_DATA (Section 19.9) frames to the sender to advertise larger limits.

A receiver can advertise a larger limit for a stream by sending a `MAX_STREAM_DATA` frame with the corresponding stream ID. A `MAX_STREAM_DATA` frame indicates the maximum absolute byte offset of a stream. A receiver could determine the flow control offset to be advertised based on the current offset of data consumed on that stream.

A receiver can advertise a larger limit for a connection by sending a `MAX_DATA` frame, which indicates the maximum of the sum of the absolute byte offsets of all streams. A receiver maintains a cumulative sum of bytes received on all streams, which is used to check for violations of the advertised connection or stream data limits. A receiver could determine the maximum data limit to be advertised based on the sum of bytes consumed on all streams.

Once a receiver advertises a limit for the connection or a stream, it is not an error to advertise a smaller limit, but the smaller limit has no effect.

A receiver **MUST** close the connection with a `FLOW_CONTROL_ERROR` error (Section 11) if the sender violates the advertised connection or stream data limits.

A sender **MUST** ignore any `MAX_STREAM_DATA` or `MAX_DATA` frames that do not increase flow control limits.

If a sender has sent data up to the limit, it will be unable to send new data and is considered blocked. A sender **SHOULD** send a `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frame to indicate to the receiver that it has data to write but is blocked by flow control limits. If a sender is blocked for a period longer than the idle timeout (Section 10.1), the receiver might close the connection even when the sender has data that is available for transmission. To keep the connection from closing, a sender that is flow control limited **SHOULD** periodically send a `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frame when it has no ack-eliciting packets in flight.

4.2. Increasing Flow Control Limits

Implementations decide when and how much credit to advertise in `MAX_STREAM_DATA` and `MAX_DATA` frames, but this section offers a few considerations.

To avoid blocking a sender, a receiver **MAY** send a `MAX_STREAM_DATA` or `MAX_DATA` frame multiple times within a round trip or send it early enough to allow time for loss of the frame and subsequent recovery.

Control frames contribute to connection overhead. Therefore, frequently sending MAX_STREAM_DATA and MAX_DATA frames with small changes is undesirable. On the other hand, if updates are less frequent, larger increments to limits are necessary to avoid blocking a sender, requiring larger resource commitments at the receiver. There is a trade-off between resource commitment and overhead when determining how large a limit is advertised.

A receiver can use an autotuning mechanism to tune the frequency and amount of advertised additional credit based on a round-trip time estimate and the rate at which the receiving application consumes data, similar to common TCP implementations. As an optimization, an endpoint could send frames related to flow control only when there are other frames to send, ensuring that flow control does not cause extra packets to be sent.

A blocked sender is not required to send STREAM_DATA_BLOCKED or DATA_BLOCKED frames. Therefore, a receiver MUST NOT wait for a STREAM_DATA_BLOCKED or DATA_BLOCKED frame before sending a MAX_STREAM_DATA or MAX_DATA frame; doing so could result in the sender being blocked for the rest of the connection. Even if the sender sends these frames, waiting for them will result in the sender being blocked for at least an entire round trip.

When a sender receives credit after being blocked, it might be able to send a large amount of data in response, resulting in short-term congestion; see Section 7.7 in [QUIC-RECOVERY] for a discussion of how a sender can avoid this congestion.

4.3. Flow Control Performance

If an endpoint cannot ensure that its peer always has available flow control credit that is greater than the peer's bandwidth-delay product on this connection, its receive throughput will be limited by flow control.

Packet loss can cause gaps in the receive buffer, preventing the application from consuming data and freeing up receive buffer space.

Sending timely updates of flow control limits can improve performance. Sending packets only to provide flow control updates can increase network load and adversely affect performance. Sending flow control updates along with other frames, such as ACK frames, reduces the cost of those updates.

4.4. Handling Stream Cancellation

Endpoints need to eventually agree on the amount of flow control credit that has been consumed on every stream, to be able to account for all bytes for connection-level flow control.

On receipt of a RESET_STREAM frame, an endpoint will tear down state for the matching stream and ignore further data arriving on that stream.

RESET_STREAM terminates one direction of a stream abruptly. For a bidirectional stream, RESET_STREAM has no effect on data flow in the opposite direction. Both endpoints MUST maintain flow control state for the stream in the unterminated direction until that direction enters a terminal state.

4.5. Stream Final Size

The final size is the amount of flow control credit that is consumed by a stream. Assuming that every contiguous byte on the stream was sent once, the final size is the number of bytes sent. More generally, this is one higher than the offset of the byte with the largest offset sent on the stream, or zero if no bytes were sent.

A sender always communicates the final size of a stream to the receiver reliably, no matter how the stream is terminated. The final size is the sum of the Offset and Length fields of a STREAM frame with a FIN flag, noting that these fields might be implicit. Alternatively, the Final Size field of a RESET_STREAM frame carries this value. This guarantees that both endpoints agree on how much flow control credit was consumed by the sender on that stream.

An endpoint will know the final size for a stream when the receiving part of the stream enters the "Size Known" or "Reset Recvd" state (Section 3). The receiver MUST use the final size of the stream to account for all bytes sent on the stream in its connection level flow controller.

An endpoint MUST NOT send data on a stream at or beyond the final size.

Once a final size for a stream is known, it cannot change. If a RESET_STREAM or STREAM frame is received indicating a change in the final size for the stream, an endpoint SHOULD respond with a FINAL_SIZE_ERROR error; see Section 11. A receiver SHOULD treat receipt of data at or beyond the final size as a FINAL_SIZE_ERROR error, even after a stream is closed. Generating these errors is not mandatory, because requiring that an endpoint generate these errors also means that the endpoint needs to maintain the final size state for closed streams, which could mean a significant state commitment.

4.6. Controlling Concurrency

An endpoint limits the cumulative number of incoming streams a peer can open. Only streams with a stream ID less than ($\text{max_stream} * 4 + \text{initial_stream_id_for_type}$) can be opened; see Table 1. Initial limits are set in the transport parameters; see Section 18.2. Subsequent limits are advertised using MAX_STREAMS frames; see Section 19.11. Separate limits apply to unidirectional and bidirectional streams.

If a max_streams transport parameter or a MAX_STREAMS frame is received with a value greater than 2^{60} , this would allow a maximum stream ID that cannot be expressed as a variable-length integer; see Section 16. If either is received, the connection MUST be closed immediately with a connection error of type TRANSPORT_PARAMETER_ERROR if the offending value was received in a transport parameter or of type FRAME_ENCODING_ERROR if it was received in a frame; see Section 10.2.

Endpoints MUST NOT exceed the limit set by their peer. An endpoint that receives a frame with a stream ID exceeding the limit it has sent MUST treat this as a connection error of type STREAM_LIMIT_ERROR (Section 11).

Once a receiver advertises a stream limit using the MAX_STREAMS frame, advertising a smaller limit has no effect. A receiver MUST ignore any MAX_STREAMS frame that does not increase the stream limit.

As with stream and connection flow control, this document leaves implementations to decide when and how many streams should be advertised to a peer via MAX_STREAMS. Implementations might choose to increase limits as streams are closed, to keep the number of streams available to peers roughly consistent.

An endpoint that is unable to open a new stream due to the peer's limits SHOULD send a STREAMS_BLOCKED frame (Section 19.14). This signal is considered useful for debugging. An endpoint MUST NOT wait to receive this signal before advertising additional credit, since

doing so will mean that the peer will be blocked for at least an entire round trip, and potentially indefinitely if the peer chooses not to send STREAMS_BLOCKED frames.

5. Connections

A QUIC connection is shared state between a client and a server.

Each connection starts with a handshake phase, during which the two endpoints establish a shared secret using the cryptographic handshake protocol [QUIC-TLS] and negotiate the application protocol. The handshake (Section 7) confirms that both endpoints are willing to communicate (Section 8.1) and establishes parameters for the connection (Section 7.4).

An application protocol can use the connection during the handshake phase with some limitations. 0-RTT allows application data to be sent by a client before receiving a response from the server. However, 0-RTT provides no protection against replay attacks; see Section 9.2 of [QUIC-TLS]. A server can also send application data to a client before it receives the final cryptographic handshake messages that allow it to confirm the identity and liveness of the client. These capabilities allow an application protocol to offer the option of trading some security guarantees for reduced latency.

The use of connection IDs (Section 5.1) allows connections to migrate to a new network path, both as a direct choice of an endpoint and when forced by a change in a middlebox. Section 9 describes mitigations for the security and privacy issues associated with migration.

For connections that are no longer needed or desired, there are several ways for a client and server to terminate a connection, as described in Section 10.

5.1. Connection ID

Each connection possesses a set of connection identifiers, or connection IDs, each of which can identify the connection. Connection IDs are independently selected by endpoints; each endpoint selects the connection IDs that its peer uses.

The primary function of a connection ID is to ensure that changes in addressing at lower protocol layers (UDP, IP) do not cause packets for a QUIC connection to be delivered to the wrong endpoint. Each endpoint selects connection IDs using an implementation-specific (and perhaps deployment-specific) method that will allow packets with that connection ID to be routed back to the endpoint and to be identified by the endpoint upon receipt.

Multiple connection IDs are used so that endpoints can send packets that cannot be identified by an observer as being for the same connection without cooperation from an endpoint; see Section 9.5.

Connection IDs **MUST NOT** contain any information that can be used by an external observer (that is, one that does not cooperate with the issuer) to correlate them with other connection IDs for the same connection. As a trivial example, this means the same connection ID **MUST NOT** be issued more than once on the same connection.

Packets with long headers include Source Connection ID and Destination Connection ID fields. These fields are used to set the connection IDs for new connections; see Section 7.2 for details.

Packets with short headers (Section 17.3) only include the Destination Connection ID and omit the explicit length. The length of the Destination Connection ID field is expected to be known to endpoints. Endpoints using a load balancer that routes based on connection ID could agree with the load balancer on a fixed length for connection IDs, or agree on an encoding scheme. A fixed portion could encode an explicit length, which allows the entire connection ID to vary in length and still be used by the load balancer.

A Version Negotiation (Section 17.2.1) packet echoes the connection IDs selected by the client, both to ensure correct routing toward the client and to demonstrate that the packet is in response to a packet sent by the client.

A zero-length connection ID can be used when a connection ID is not needed to route to the correct endpoint. However, multiplexing connections on the same local IP address and port while using zero-length connection IDs will cause failures in the presence of peer connection migration, NAT rebinding, and client port reuse. An endpoint **MUST NOT** use the same IP address and port for multiple concurrent connections with zero-length connection IDs, unless it is certain that those protocol features are not in use.

When an endpoint uses a non-zero-length connection ID, it needs to ensure that the peer has a supply of connection IDs from which to choose for packets sent to the endpoint. These connection IDs are supplied by the endpoint using the `NEW_CONNECTION_ID` frame (Section 19.15).

5.1.1. Issuing Connection IDs

Each Connection ID has an associated sequence number to assist in detecting when `NEW_CONNECTION_ID` or `RETIRE_CONNECTION_ID` frames refer to the same value. The initial connection ID issued by an endpoint is sent in the Source Connection ID field of the long packet header (Section 17.2) during the handshake. The sequence number of the initial connection ID is 0. If the `preferred_address` transport parameter is sent, the sequence number of the supplied connection ID is 1.

Additional connection IDs are communicated to the peer using `NEW_CONNECTION_ID` frames (Section 19.15). The sequence number on each newly issued connection ID **MUST** increase by 1. The connection ID that a client selects for the first Destination Connection ID field it sends and any connection ID provided by a Retry packet are not assigned sequence numbers.

When an endpoint issues a connection ID, it **MUST** accept packets that carry this connection ID for the duration of the connection or until its peer invalidates the connection ID via a `RETIRE_CONNECTION_ID` frame (Section 19.16). Connection IDs that are issued and not retired are considered active; any active connection ID is valid for use with the current connection at any time, in any packet type. This includes the connection ID issued by the server via the `preferred_address` transport parameter.

An endpoint **SHOULD** ensure that its peer has a sufficient number of available and unused connection IDs. Endpoints advertise the number of active connection IDs they are willing to maintain using the `active_connection_id_limit` transport parameter. An endpoint **MUST NOT** provide more connection IDs than the peer's limit. An endpoint **MAY** send connection IDs that temporarily exceed a peer's limit if the `NEW_CONNECTION_ID` frame also requires the retirement of any excess, by including a sufficiently large value in the Retire Prior To field.

A `NEW_CONNECTION_ID` frame might cause an endpoint to add some active connection IDs and retire others based on the value of the `Retire Prior To` field. After processing a `NEW_CONNECTION_ID` frame and adding and retiring active connection IDs, if the number of active connection IDs exceeds the value advertised in its `active_connection_id_limit` transport parameter, an endpoint **MUST** close the connection with an error of type `CONNECTION_ID_LIMIT_ERROR`.

An endpoint **SHOULD** supply a new connection ID when the peer retires a connection ID. If an endpoint provided fewer connection IDs than the peer's `active_connection_id_limit`, it **MAY** supply a new connection ID when it receives a packet with a previously unused connection ID. An endpoint **MAY** limit the total number of connection IDs issued for each connection to avoid the risk of running out of connection IDs; see Section 10.3.2. An endpoint **MAY** also limit the issuance of connection IDs to reduce the amount of per-path state it maintains, such as path validation status, as its peer might interact with it over as many paths as there are issued connection IDs.

An endpoint that initiates migration and requires non-zero-length connection IDs **SHOULD** ensure that the pool of connection IDs available to its peer allows the peer to use a new connection ID on migration, as the peer will be unable to respond if the pool is exhausted.

An endpoint that selects a zero-length connection ID during the handshake cannot issue a new connection ID. A zero-length Destination Connection ID field is used in all packets sent toward such an endpoint over any network path.

5.1.2. Consuming and Retiring Connection IDs

An endpoint can change the connection ID it uses for a peer to another available one at any time during the connection. An endpoint consumes connection IDs in response to a migrating peer; see Section 9.5 for more.

An endpoint maintains a set of connection IDs received from its peer, any of which it can use when sending packets. When the endpoint wishes to remove a connection ID from use, it sends a `RETIRE_CONNECTION_ID` frame to its peer. Sending a `RETIRE_CONNECTION_ID` frame indicates that the connection ID will not be used again and requests that the peer replace it with a new connection ID using a `NEW_CONNECTION_ID` frame.

As discussed in Section 9.5, endpoints limit the use of a connection ID to packets sent from a single local address to a single destination address. Endpoints SHOULD retire connection IDs when they are no longer actively using either the local or destination address for which the connection ID was used.

An endpoint might need to stop accepting previously issued connection IDs in certain circumstances. Such an endpoint can cause its peer to retire connection IDs by sending a NEW_CONNECTION_ID frame with an increased Retire Prior To field. The endpoint SHOULD continue to accept the previously issued connection IDs until they are retired by the peer. If the endpoint can no longer process the indicated connection IDs, it MAY close the connection.

Upon receipt of an increased Retire Prior To field, the peer MUST stop using the corresponding connection IDs and retire them with RETIRE_CONNECTION_ID frames before adding the newly provided connection ID to the set of active connection IDs. This ordering allows an endpoint to replace all active connection IDs without the possibility of a peer having no available connection IDs and without exceeding the limit the peer sets in the active_connection_id_limit transport parameter; see Section 18.2. Failure to cease using the connection IDs when requested can result in connection failures, as the issuing endpoint might be unable to continue using the connection IDs with the active connection.

An endpoint SHOULD limit the number of connection IDs it has retired locally and have not yet been acknowledged. An endpoint SHOULD allow for sending and tracking a number of RETIRE_CONNECTION_ID frames of at least twice the active_connection_id limit. An endpoint MUST NOT forget a connection ID without retiring it, though it MAY choose to treat having connection IDs in need of retirement that exceed this limit as a connection error of type CONNECTION_ID_LIMIT_ERROR.

Endpoints SHOULD NOT issue updates of the Retire Prior To field before receiving RETIRE_CONNECTION_ID frames that retire all connection IDs indicated by the previous Retire Prior To value.

5.2. Matching Packets to Connections

Incoming packets are classified on receipt. Packets can either be associated with an existing connection, or - for servers - potentially create a new connection.

Endpoints try to associate a packet with an existing connection. If the packet has a non-zero-length Destination Connection ID corresponding to an existing connection, QUIC processes that packet accordingly. Note that more than one connection ID can be associated with a connection; see Section 5.1.

If the Destination Connection ID is zero length and the addressing information in the packet matches the addressing information the endpoint uses to identify a connection with a zero-length connection ID, QUIC processes the packet as part of that connection. An endpoint can use just destination IP and port or both source and destination addresses for identification, though this makes connections fragile as described in Section 5.1.

Endpoints can send a Stateless Reset (Section 10.3) for any packets that cannot be attributed to an existing connection. A stateless reset allows a peer to more quickly identify when a connection becomes unusable.

Packets that are matched to an existing connection are discarded if the packets are inconsistent with the state of that connection. For example, packets are discarded if they indicate a different protocol version than that of the connection, or if the removal of packet protection is unsuccessful once the expected keys are available.

Invalid packets that lack strong integrity protection, such as Initial, Retry, or Version Negotiation, MAY be discarded. An endpoint MUST generate a connection error if processing the contents of these packets prior to discovering an error, or fully revert any changes made during that processing.

5.2.1. Client Packet Handling

Valid packets sent to clients always include a Destination Connection ID that matches a value the client selects. Clients that choose to receive zero-length connection IDs can use the local address and port to identify a connection. Packets that do not match an existing connection, based on Destination Connection ID or, if this value is zero-length, local IP address and port, are discarded.

Due to packet reordering or loss, a client might receive packets for a connection that are encrypted with a key it has not yet computed. The client MAY drop these packets, or MAY buffer them in anticipation of later packets that allow it to compute the key.

If a client receives a packet that uses a different version than it initially selected, it MUST discard that packet.

5.2.2. Server Packet Handling

If a server receives a packet that indicates an unsupported version and if the packet is large enough to initiate a new connection for any supported version, the server **SHOULD** send a Version Negotiation packet as described in Section 6.1. A server **MAY** limit the number of packets to which it responds with a Version Negotiation packet. Servers **MUST** drop smaller packets that specify unsupported versions.

The first packet for an unsupported version can use different semantics and encodings for any version-specific field. In particular, different packet protection keys might be used for different versions. Servers that do not support a particular version are unlikely to be able to decrypt the payload of the packet or properly interpret the result. Servers **SHOULD** respond with a Version Negotiation packet, provided that the datagram is sufficiently long.

Packets with a supported version, or no version field, are matched to a connection using the connection ID or - for packets with zero-length connection IDs - the local address and port. These packets are processed using the selected connection; otherwise, the server continues below.

If the packet is an Initial packet fully conforming with the specification, the server proceeds with the handshake (Section 7). This commits the server to the version that the client selected.

If a server refuses to accept a new connection, it **SHOULD** send an Initial packet containing a `CONNECTION_CLOSE` frame with error code `CONNECTION_REFUSED`.

If the packet is a 0-RTT packet, the server **MAY** buffer a limited number of these packets in anticipation of a late-arriving Initial packet. Clients are not able to send Handshake packets prior to receiving a server response, so servers **SHOULD** ignore any such packets.

Servers **MUST** drop incoming packets under all other circumstances.

5.2.3. Considerations for Simple Load Balancers

A server deployment could load balance among servers using only source and destination IP addresses and ports. Changes to the client's IP address or port could result in packets being forwarded to the wrong server. Such a server deployment could use one of the following methods for connection continuity when a client's address changes.

- * Servers could use an out-of-band mechanism to forward packets to the correct server based on Connection ID.
- * If servers can use a dedicated server IP address or port, other than the one that the client initially connects to, they could use the `preferred_address` transport parameter to request that clients move connections to that dedicated address. Note that clients could choose not to use the preferred address.

A server in a deployment that does not implement a solution to maintain connection continuity when the client address changes **SHOULD** indicate migration is not supported using the `disable_active_migration` transport parameter. The `disable_active_migration` transport parameter does not prohibit connection migration after a client has acted on a `preferred_address` transport parameter.

Server deployments that use this simple form of load balancing **MUST** avoid the creation of a stateless reset oracle; see Section 21.11.

5.3. Operations on Connections

This document does not define an API for QUIC, but instead defines a set of functions for QUIC connections that application protocols can rely upon. An application protocol can assume that an implementation of QUIC provides an interface that includes the operations described in this section. An implementation designed for use with a specific application protocol might provide only those operations that are used by that protocol.

When implementing the client role, an application protocol can:

- * open a connection, which begins the exchange described in Section 7;
- * enable Early Data when available; and
- * be informed when Early Data has been accepted or rejected by a server.

When implementing the server role, an application protocol can:

- * listen for incoming connections, which prepares for the exchange described in Section 7;
- * if Early Data is supported, embed application-controlled data in the TLS resumption ticket sent to the client; and

- * if Early Data is supported, retrieve application-controlled data from the client's resumption ticket and accept or reject Early Data based on that information.

In either role, an application protocol can:

- * configure minimum values for the initial number of permitted streams of each type, as communicated in the transport parameters (Section 7.4);
- * control resource allocation for receive buffers by setting flow control limits both for streams and for the connection
- * identify whether the handshake has completed successfully or is still ongoing;
- * keep a connection from silently closing, either by generating PING frames (Section 19.2) or by requesting that the transport send additional frames before the idle timeout expires (Section 10.1); and
- * immediately close (Section 10.2) the connection.

6. Version Negotiation

Version negotiation allows a server to indicate that it does not support the version the client used. A server sends a Version Negotiation packet in response to each packet that might initiate a new connection; see Section 5.2 for details.

The size of the first packet sent by a client will determine whether a server sends a Version Negotiation packet. Clients that support multiple QUIC versions SHOULD ensure that the first UDP datagram they send is sized to the largest of the minimum datagram sizes from all versions they support, using PADDING frames (Section 19.1) as necessary. This ensures that the server responds if there is a mutually supported version. A server might not send a Version Negotiation packet if the datagram it receives is smaller than the minimum size specified in a different version; see Section 14.1.

6.1. Sending Version Negotiation Packets

If the version selected by the client is not acceptable to the server, the server responds with a Version Negotiation packet; see Section 17.2.1. This includes a list of versions that the server will accept. An endpoint MUST NOT send a Version Negotiation packet in response to receiving a Version Negotiation packet.

This system allows a server to process packets with unsupported versions without retaining state. Though either the Initial packet or the Version Negotiation packet that is sent in response could be lost, the client will send new packets until it successfully receives a response or it abandons the connection attempt.

A server MAY limit the number of Version Negotiation packets it sends. For instance, a server that is able to recognize packets as 0-RTT might choose not to send Version Negotiation packets in response to 0-RTT packets with the expectation that it will eventually receive an Initial packet.

6.2. Handling Version Negotiation Packets

Version Negotiation packets are designed to allow for functionality to be defined in the future that allows QUIC to negotiate the version of QUIC to use for a connection. Future standards-track specifications might change how implementations that support multiple versions of QUIC react to Version Negotiation packets received in response to an attempt to establish a connection using this version.

A client that supports only this version of QUIC MUST abandon the current connection attempt if it receives a Version Negotiation packet, with the following two exceptions. A client MUST discard any Version Negotiation packet if it has received and successfully processed any other packet, including an earlier Version Negotiation packet. A client MUST discard a Version Negotiation packet that lists the QUIC version selected by the client.

How to perform version negotiation is left as future work defined by future standards-track specifications. In particular, that future work will ensure robustness against version downgrade attacks; see Section 21.12.

6.2.1. Version Negotiation Between Draft Versions

[[RFC editor: please remove this section before publication.]]

When a draft implementation receives a Version Negotiation packet, it MAY use it to attempt a new connection with one of the versions listed in the packet, instead of abandoning the current connection attempt; see Section 6.2.

The client MUST check that the Destination and Source Connection ID fields match the Source and Destination Connection ID fields in a packet that the client sent. If this check fails, the packet MUST be discarded.

Once the Version Negotiation packet is determined to be valid, the client then selects an acceptable protocol version from the list provided by the server. The client then attempts to create a new connection using that version. The new connection **MUST** use a new random Destination Connection ID different from the one it had previously sent.

Note that this mechanism does not protect against downgrade attacks and **MUST NOT** be used outside of draft implementations.

6.3. Using Reserved Versions

For a server to use a new version in the future, clients need to correctly handle unsupported versions. Some version numbers (0x?a?a?a as defined in Section 15) are reserved for inclusion in fields that contain version numbers.

Endpoints **MAY** add reserved versions to any field where unknown or unsupported versions are ignored to test that a peer correctly ignores the value. For instance, an endpoint could include a reserved version in a Version Negotiation packet; see Section 17.2.1. Endpoints **MAY** send packets with a reserved version to test that a peer correctly discards the packet.

7. Cryptographic and Transport Handshake

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC uses the CRYPTO frame (Section 19.6) to transmit the cryptographic handshake. The version of QUIC defined in this document is identified as 0x00000001 and uses TLS as described in [QUIC-TLS]; a different QUIC version could indicate that a different cryptographic handshake protocol is in use.

QUIC provides reliable, ordered delivery of the cryptographic handshake data. QUIC packet protection is used to encrypt as much of the handshake protocol as possible. The cryptographic handshake **MUST** provide the following properties:

- * authenticated key exchange, where
 - a server is always authenticated,
 - a client is optionally authenticated,
 - every connection produces distinct and unrelated keys, and

- keying material is usable for packet protection for both 0-RTT and 1-RTT packets
- * authenticated exchange of values for transport parameters of both endpoints, and confidentiality protection for server transport parameters (see Section 7.4)
- * authenticated negotiation of an application protocol (TLS uses ALPN [ALPN] for this purpose)

The CRYPTO frame can be sent in different packet number spaces (Section 12.3). The offsets used by CRYPTO frames to ensure ordered delivery of cryptographic handshake data start from zero in each packet number space.

Figure 4 shows a simplified handshake and the exchange of packets and frames that are used to advance the handshake. Exchange of application data during the handshake is enabled where possible, shown with a '*'. Once the handshake is complete, endpoints are able to exchange application data freely.

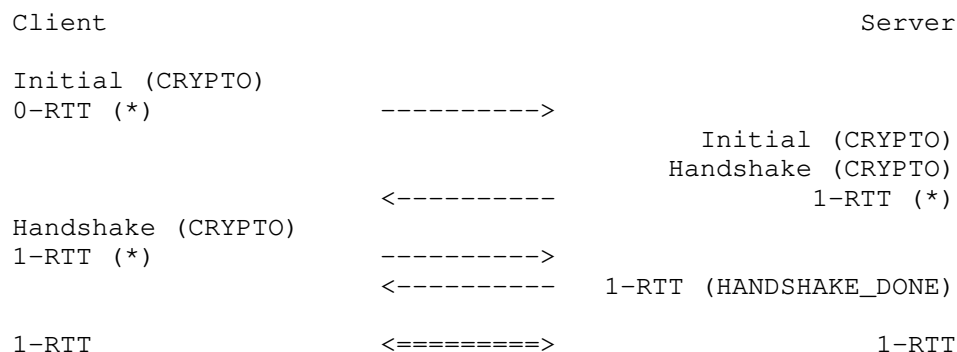


Figure 4: Simplified QUIC Handshake

Endpoints can use packets sent during the handshake to test for Explicit Congestion Notification (ECN) support; see Section 13.4. An endpoint validates support for ECN by observing whether the ACK frames acknowledging the first packets it sends carry ECN counts, as described in Section 13.4.2.

Endpoints MUST explicitly negotiate an application protocol. This avoids situations where there is a disagreement about the protocol that is in use.

7.1. Example Handshake Flows

Details of how TLS is integrated with QUIC are provided in [QUIC-TLS], but some examples are provided here. An extension of this exchange to support client address validation is shown in Section 8.1.2.

Once any address validation exchanges are complete, the cryptographic handshake is used to agree on cryptographic keys. The cryptographic handshake is carried in Initial (Section 17.2.2) and Handshake (Section 17.2.4) packets.

Figure 5 provides an overview of the 1-RTT handshake. Each line shows a QUIC packet with the packet type and packet number shown first, followed by the frames that are typically contained in those packets. So, for instance the first packet is of type Initial, with packet number 0, and contains a CRYPTO frame carrying the ClientHello.

Multiple QUIC packets -- even of different packet types -- can be coalesced into a single UDP datagram; see Section 12.2. As a result, this handshake could consist of as few as 4 UDP datagrams, or any number more (subject to limits inherent to the protocol, such as congestion control and anti-amplification). For instance, the server's first flight contains Initial packets, Handshake packets, and "0.5-RTT data" in 1-RTT packets.

Client	Server
Initial[0]: CRYPTO[CH] ->	
	Initial[0]: CRYPTO[SH] ACK[0]
	Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
	<- 1-RTT[0]: STREAM[1, "..."]
Initial[1]: ACK[0]	
Handshake[0]: CRYPTO[FIN], ACK[0]	
1-RTT[0]: STREAM[0, "..."], ACK[0] ->	
	Handshake[1]: ACK[0]
	<- 1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[0]

Figure 5: Example 1-RTT Handshake

Figure 6 shows an example of a connection with a 0-RTT handshake and a single packet of 0-RTT data. Note that as described in Section 12.3, the server acknowledges 0-RTT data in 1-RTT packets, and the client sends 1-RTT packets in the same packet number space.

Client

Server

```

Initial[0]: CRYPTO[CH]
0-RTT[0]: STREAM[0, "..."] ->

                                Initial[0]: CRYPTO[SH] ACK[0]
                                Handshake[0] CRYPTO[EE, FIN]
                                <- 1-RTT[0]: STREAM[1, "..."] ACK[0]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[1]: STREAM[0, "..."] ACK[0] ->

                                Handshake[1]: ACK[0]
                                <- 1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[1]

```

Figure 6: Example 0-RTT Handshake

7.2. Negotiating Connection IDs

A connection ID is used to ensure consistent routing of packets, as described in Section 5.1. The long header contains two connection IDs: the Destination Connection ID is chosen by the recipient of the packet and is used to provide consistent routing; the Source Connection ID is used to set the Destination Connection ID used by the peer.

During the handshake, packets with the long header (Section 17.2) are used to establish the connection IDs used by both endpoints. Each endpoint uses the Source Connection ID field to specify the connection ID that is used in the Destination Connection ID field of packets being sent to them. After processing the first Initial packet, each endpoint sets the Destination Connection ID field in subsequent packets it sends to the value of the Source Connection ID field that it received.

When an Initial packet is sent by a client that has not previously received an Initial or Retry packet from the server, the client populates the Destination Connection ID field with an unpredictable value. This Destination Connection ID MUST be at least 8 bytes in length. Until a packet is received from the server, the client MUST use the same Destination Connection ID value on all packets in this connection.

The Destination Connection ID field from the first Initial packet sent by a client is used to determine packet protection keys for Initial packets. These keys change after receiving a Retry packet; see Section 5.2 of [QUIC-TLS].

The client populates the Source Connection ID field with a value of its choosing and sets the Source Connection ID Length field to indicate the length.

The first flight of 0-RTT packets use the same Destination Connection ID and Source Connection ID values as the client's first Initial packet.

Upon first receiving an Initial or Retry packet from the server, the client uses the Source Connection ID supplied by the server as the Destination Connection ID for subsequent packets, including any 0-RTT packets. This means that a client might have to change the connection ID it sets in the Destination Connection ID field twice during connection establishment: once in response to a Retry, and once in response to an Initial packet from the server. Once a client has received a valid Initial packet from the server, it **MUST** discard any subsequent packet it receives on that connection with a different Source Connection ID.

A client **MUST** change the Destination Connection ID it uses for sending packets in response to only the first received Initial or Retry packet. A server **MUST** set the Destination Connection ID it uses for sending packets based on the first received Initial packet. Any further changes to the Destination Connection ID are only permitted if the values are taken from `NEW_CONNECTION_ID` frames; if subsequent Initial packets include a different Source Connection ID, they **MUST** be discarded. This avoids unpredictable outcomes that might otherwise result from stateless processing of multiple Initial packets with different Source Connection IDs.

The Destination Connection ID that an endpoint sends can change over the lifetime of a connection, especially in response to connection migration (Section 9); see Section 5.1.1 for details.

7.3. Authenticating Connection IDs

The choice each endpoint makes about connection IDs during the handshake is authenticated by including all values in transport parameters; see Section 7.4. This ensures that all connection IDs used for the handshake are also authenticated by the cryptographic handshake.

Each endpoint includes the value of the Source Connection ID field from the first Initial packet it sent in the `initial_source_connection_id` transport parameter; see Section 18.2. A server includes the Destination Connection ID field from the first Initial packet it received from the client in the `original_destination_connection_id` transport parameter; if the server

sent a Retry packet, this refers to the first Initial packet received before sending the Retry packet. If it sends a Retry packet, a server also includes the Source Connection ID field from the Retry packet in the `retry_source_connection_id` transport parameter.

The values provided by a peer for these transport parameters **MUST** match the values that an endpoint used in the Destination and Source Connection ID fields of Initial packets that it sent (and received, for servers). Endpoints **MUST** validate that received transport parameters match received Connection ID values. Including connection ID values in transport parameters and verifying them ensures that that an attacker cannot influence the choice of connection ID for a successful connection by injecting packets carrying attacker-chosen connection IDs during the handshake.

An endpoint **MUST** treat absence of the `initial_source_connection_id` transport parameter from either endpoint or absence of the `original_destination_connection_id` transport parameter from the server as a connection error of type `TRANSPORT_PARAMETER_ERROR`.

An endpoint **MUST** treat the following as a connection error of type `TRANSPORT_PARAMETER_ERROR` or `PROTOCOL_VIOLATION`:

- * absence of the `retry_source_connection_id` transport parameter from the server after receiving a Retry packet,
- * presence of the `retry_source_connection_id` transport parameter when no Retry packet was received, or
- * a mismatch between values received from a peer in these transport parameters and the value sent in the corresponding Destination or Source Connection ID fields of Initial packets.

If a zero-length connection ID is selected, the corresponding transport parameter is included with a zero-length value.

Figure 7 shows the connection IDs (with DCID=Destination Connection ID, SCID=Source Connection ID) that are used in a complete handshake. The exchange of Initial packets is shown, plus the later exchange of 1-RTT packets that includes the connection ID established during the handshake.

Client	Server
Initial: DCID=S1, SCID=C1 ->	
	<- Initial: DCID=C1, SCID=S3
	...
1-RTT: DCID=S3 ->	
	<- 1-RTT: DCID=C1

Figure 7: Use of Connection IDs in a Handshake

Figure 8 shows a similar handshake that includes a Retry packet.

Client	Server
Initial: DCID=S1, SCID=C1 ->	
	<- Retry: DCID=C1, SCID=S2
Initial: DCID=S2, SCID=C1 ->	
	<- Initial: DCID=C1, SCID=S3
	...
1-RTT: DCID=S3 ->	
	<- 1-RTT: DCID=C1

Figure 8: Use of Connection IDs in a Handshake with Retry

In both cases (Figure 7 and Figure 8), the client sets the value of the `initial_source_connection_id` transport parameter to "C1".

When the handshake does not include a Retry (Figure 7), the server sets `original_destination_connection_id` to "S1" and `initial_source_connection_id` to "S3". In this case, the server does not include a `retry_source_connection_id` transport parameter.

When the handshake includes a Retry (Figure 8), the server sets `original_destination_connection_id` to "S1", `retry_source_connection_id` to "S2", and `initial_source_connection_id` to "S3".

7.4. Transport Parameters

During connection establishment, both endpoints make authenticated declarations of their transport parameters. Endpoints are required to comply with the restrictions that each parameter defines; the description of each parameter includes rules for its handling.

Transport parameters are declarations that are made unilaterally by each endpoint. Each endpoint can choose values for transport parameters independent of the values chosen by its peer.

The encoding of the transport parameters is detailed in Section 18.

QUIC includes the encoded transport parameters in the cryptographic handshake. Once the handshake completes, the transport parameters declared by the peer are available. Each endpoint validates the values provided by its peer.

Definitions for each of the defined transport parameters are included in Section 18.2.

An endpoint **MUST** treat receipt of a transport parameter with an invalid value as a connection error of type `TRANSPORT_PARAMETER_ERROR`.

An endpoint **MUST NOT** send a parameter more than once in a given transport parameters extension. An endpoint **SHOULD** treat receipt of duplicate transport parameters as a connection error of type `TRANSPORT_PARAMETER_ERROR`.

Endpoints use transport parameters to authenticate the negotiation of connection IDs during the handshake; see Section 7.3.

Application Layer Protocol Negotiation (ALPN; see [ALPN]) allows clients to offer multiple application protocols during connection establishment. The transport parameters that a client includes during the handshake apply to all application protocols that the client offers. Application protocols can recommend values for transport parameters, such as the initial flow control limits. However, application protocols that set constraints on values for transport parameters could make it impossible for a client to offer multiple application protocols if these constraints conflict.

7.4.1. Values of Transport Parameters for 0-RTT

Using 0-RTT depends on both client and server using protocol parameters that were negotiated from a previous connection. To enable 0-RTT, endpoints store the value of the server transport parameters from a connection and apply them to any 0-RTT packets that are sent in subsequent connections to that peer that use a session ticket issued on that connection. This information is stored with any information required by the application protocol or cryptographic handshake; see Section 4.6 of [QUIC-TLS].

Remembered transport parameters apply to the new connection until the handshake completes and the client starts sending 1-RTT packets. Once the handshake completes, the client uses the transport parameters established in the handshake. Not all transport parameters are remembered, as some do not apply to future connections or they have no effect on use of 0-RTT.

The definition of a new transport parameter (Section 7.4.2) MUST specify whether storing the transport parameter for 0-RTT is mandatory, optional, or prohibited. A client need not store a transport parameter it cannot process.

A client MUST NOT use remembered values for the following parameters: `ack_delay_exponent`, `max_ack_delay`, `initial_source_connection_id`, `original_destination_connection_id`, `preferred_address`, `retry_source_connection_id`, and `stateless_reset_token`. The client MUST use the server's new values in the handshake instead; if the server does not provide new values, the default value is used.

A client that attempts to send 0-RTT data MUST remember all other transport parameters used by the server that it is able to process. The server can remember these transport parameters, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the transport parameters in determining whether to accept 0-RTT data.

If 0-RTT data is accepted by the server, the server MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. In particular, a server that accepts 0-RTT data MUST NOT set values for the following parameters (Section 18.2) that are smaller than the remembered value of the parameters.

- * `active_connection_id_limit`
- * `initial_max_data`
- * `initial_max_stream_data_bidi_local`
- * `initial_max_stream_data_bidi_remote`
- * `initial_max_stream_data_uni`
- * `initial_max_streams_bidi`
- * `initial_max_streams_uni`

Omitting or setting a zero value for certain transport parameters can result in 0-RTT data being enabled, but not usable. The applicable subset of transport parameters that permit sending of application data SHOULD be set to non-zero values for 0-RTT. This includes `initial_max_data` and either `initial_max_streams_bidi` and `initial_max_stream_data_bidi_remote`, or `initial_max_streams_uni` and `initial_max_stream_data_uni`.

A server MAY store and recover the previously sent values of the `max_idle_timeout`, `max_udp_payload_size`, and `disable_active_migration` parameters and reject 0-RTT if it selects smaller values. Lowering the values of these parameters while also accepting 0-RTT data could degrade the performance of the connection. Specifically, lowering the `max_udp_payload_size` could result in dropped packets leading to worse performance compared to rejecting 0-RTT data outright.

A server MUST reject 0-RTT data if the restored values for transport parameters cannot be supported.

When sending frames in 0-RTT packets, a client MUST only use remembered transport parameters; importantly, it MUST NOT use updated values that it learns from the server's updated transport parameters or from frames received in 1-RTT packets. Updated values of transport parameters from the handshake apply only to 1-RTT packets. For instance, flow control limits from remembered transport parameters apply to all 0-RTT packets even if those values are increased by the handshake or by frames sent in 1-RTT packets. A server MAY treat use of updated transport parameters in 0-RTT as a connection error of type `PROTOCOL_VIOLATION`.

7.4.2. New Transport Parameters

New transport parameters can be used to negotiate new protocol behavior. An endpoint MUST ignore transport parameters that it does not support. Absence of a transport parameter therefore disables any optional protocol feature that is negotiated using the parameter. As described in Section 18.1, some identifiers are reserved in order to exercise this requirement.

A client that does not understand a transport parameter can discard it and attempt 0-RTT on subsequent connections. However, if the client adds support for a discarded transport parameter, it risks violating the constraints that the transport parameter establishes if it attempts 0-RTT. New transport parameters can avoid this problem by setting a default of the most conservative value. Clients can avoid this problem by remembering all parameters, even ones not currently supported.

New transport parameters can be registered according to the rules in Section 22.3.

7.5. Cryptographic Message Buffering

Implementations need to maintain a buffer of CRYPTO data received out of order. Because there is no flow control of CRYPTO frames, an endpoint could potentially force its peer to buffer an unbounded amount of data.

Implementations **MUST** support buffering at least 4096 bytes of data received in out-of-order CRYPTO frames. Endpoints **MAY** choose to allow more data to be buffered during the handshake. A larger limit during the handshake could allow for larger keys or credentials to be exchanged. An endpoint's buffer size does not need to remain constant during the life of the connection.

Being unable to buffer CRYPTO frames during the handshake can lead to a connection failure. If an endpoint's buffer is exceeded during the handshake, it can expand its buffer temporarily to complete the handshake. If an endpoint does not expand its buffer, it **MUST** close the connection with a CRYPTO_BUFFER_EXCEEDED error code.

Once the handshake completes, if an endpoint is unable to buffer all data in a CRYPTO frame, it **MAY** discard that CRYPTO frame and all CRYPTO frames received in the future, or it **MAY** close the connection with a CRYPTO_BUFFER_EXCEEDED error code. Packets containing discarded CRYPTO frames **MUST** be acknowledged because the packet has been received and processed by the transport even though the CRYPTO frame was discarded.

8. Address Validation

Address validation ensures that an endpoint cannot be used for a traffic amplification attack. In such an attack, a packet is sent to a server with spoofed source address information that identifies a victim. If a server generates more or larger packets in response to that packet, the attacker can use the server to send more data toward the victim than it would be able to send on its own.

The primary defense against amplification attacks is verifying that a peer is able to receive packets at the transport address that it claims. Therefore, after receiving packets from an address that is not yet validated, an endpoint **MUST** limit the amount of data it sends to the unvalidated address to three times the amount of data received from that address. This limit on the size of responses is known as the anti-amplification limit.

Address validation is performed both during connection establishment (see Section 8.1) and during connection migration (see Section 8.2).

8.1. Address Validation During Connection Establishment

Connection establishment implicitly provides address validation for both endpoints. In particular, receipt of a packet protected with Handshake keys confirms that the peer successfully processed an Initial packet. Once an endpoint has successfully processed a Handshake packet from the peer, it can consider the peer address to have been validated.

Additionally, an endpoint MAY consider the peer address validated if the peer uses a connection ID chosen by the endpoint and the connection ID contains at least 64 bits of entropy.

For the client, the value of the Destination Connection ID field in its first Initial packet allows it to validate the server address as a part of successfully processing any packet. Initial packets from the server are protected with keys that are derived from this value (see Section 5.2 of [QUIC-TLS]). Alternatively, the value is echoed by the server in Version Negotiation packets (Section 6) or included in the Integrity Tag in Retry packets (Section 5.8 of [QUIC-TLS]).

Prior to validating the client address, servers MUST NOT send more than three times as many bytes as the number of bytes they have received. This limits the magnitude of any amplification attack that can be mounted using spoofed source addresses. For the purposes of avoiding amplification prior to address validation, servers MUST count all of the payload bytes received in datagrams that are uniquely attributed to a single connection. This includes datagrams that contain packets that are successfully processed and datagrams that contain packets that are all discarded.

Clients MUST ensure that UDP datagrams containing Initial packets have UDP payloads of at least 1200 bytes, adding PADDING frames as necessary. A client that sends padded datagrams allows the server to send more data prior to completing address validation.

Loss of an Initial or Handshake packet from the server can cause a deadlock if the client does not send additional Initial or Handshake packets. A deadlock could occur when the server reaches its anti-amplification limit and the client has received acknowledgments for all the data it has sent. In this case, when the client has no reason to send additional packets, the server will be unable to send more data because it has not validated the client's address. To prevent this deadlock, clients MUST send a packet on a probe timeout (PTO, see Section 6.2 of [QUIC-RECOVERY]). Specifically, the client

MUST send an Initial packet in a UDP datagram that contains at least 1200 bytes if it does not have Handshake keys, and otherwise send a Handshake packet.

A server might wish to validate the client address before starting the cryptographic handshake. QUIC uses a token in the Initial packet to provide address validation prior to completing the handshake. This token is delivered to the client during connection establishment with a Retry packet (see Section 8.1.2) or in a previous connection using the NEW_TOKEN frame (see Section 8.1.3).

In addition to sending limits imposed prior to address validation, servers are also constrained in what they can send by the limits set by the congestion controller. Clients are only constrained by the congestion controller.

8.1.1. Token Construction

A token sent in a NEW_TOKEN frame or a Retry packet MUST be constructed in a way that allows the server to identify how it was provided to a client. These tokens are carried in the same field, but require different handling from servers.

8.1.2. Address Validation using Retry Packets

Upon receiving the client's Initial packet, the server can request address validation by sending a Retry packet (Section 17.2.5) containing a token. This token MUST be repeated by the client in all Initial packets it sends for that connection after it receives the Retry packet.

In response to processing an Initial containing a token that was provided in a Retry packet, a server cannot send another Retry packet; it can only refuse the connection or permit it to proceed.

As long as it is not possible for an attacker to generate a valid token for its own address (see Section 8.1.4) and the client is able to return that token, it proves to the server that it received the token.

A server can also use a Retry packet to defer the state and processing costs of connection establishment. Requiring the server to provide a different connection ID, along with the `original_destination_connection_id` transport parameter defined in Section 18.2, forces the server to demonstrate that it, or an entity it cooperates with, received the original Initial packet from the client. Providing a different connection ID also grants a server some control over how subsequent packets are routed. This can be used to direct connections to a different server instance.

If a server receives a client Initial that contains an invalid Retry token but is otherwise valid, it knows the client will not accept another Retry token. The server can discard such a packet and allow the client to time out to detect handshake failure, but that could impose a significant latency penalty on the client. Instead, the server **SHOULD** immediately close (Section 10.2) the connection with an `INVALID_TOKEN` error. Note that a server has not established any state for the connection at this point and so does not enter the closing period.

A flow showing the use of a Retry packet is shown in Figure 9.

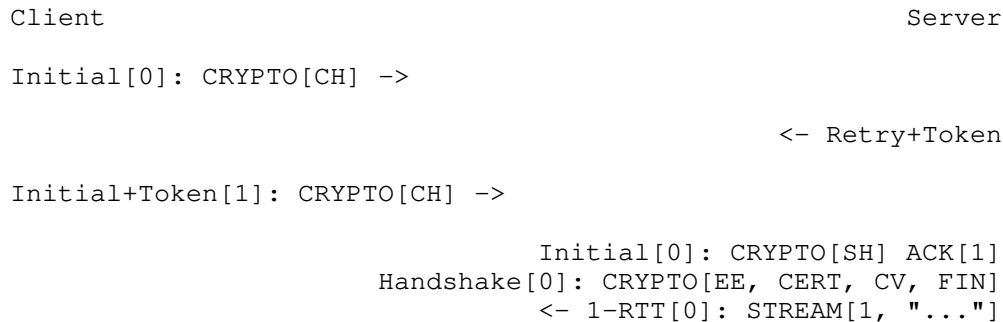


Figure 9: Example Handshake with Retry

8.1.3. Address Validation for Future Connections

A server **MAY** provide clients with an address validation token during one connection that can be used on a subsequent connection. Address validation is especially important with 0-RTT because a server potentially sends a significant amount of data to a client in response to 0-RTT data.

The server uses the `NEW_TOKEN` frame (Section 19.7) to provide the client with an address validation token that can be used to validate future connections. In a future connection, the client includes this token in Initial packets to provide address validation. The client

MUST include the token in all Initial packets it sends, unless a Retry replaces the token with a newer one. The client MUST NOT use the token provided in a Retry for future connections. Servers MAY discard any Initial packet that does not carry the expected token.

Unlike the token that is created for a Retry packet, which is used immediately, the token sent in the NEW_TOKEN frame can be used after some period of time has passed. Thus, a token SHOULD have an expiration time, which could be either an explicit expiration time or an issued timestamp that can be used to dynamically calculate the expiration time. A server can store the expiration time or include it in an encrypted form in the token.

A token issued with NEW_TOKEN MUST NOT include information that would allow values to be linked by an observer to the connection on which it was issued. For example, it cannot include the previous connection ID or addressing information, unless the values are encrypted. A server MUST ensure that every NEW_TOKEN frame it sends is unique across all clients, with the exception of those sent to repair losses of previously sent NEW_TOKEN frames. Information that allows the server to distinguish between tokens from Retry and NEW_TOKEN MAY be accessible to entities other than the server.

It is unlikely that the client port number is the same on two different connections; validating the port is therefore unlikely to be successful.

A token received in a NEW_TOKEN frame is applicable to any server that the connection is considered authoritative for (e.g., server names included in the certificate). When connecting to a server for which the client retains an applicable and unused token, it SHOULD include that token in the Token field of its Initial packet. Including a token might allow the server to validate the client address without an additional round trip. A client MUST NOT include a token that is not applicable to the server that it is connecting to, unless the client has the knowledge that the server that issued the token and the server the client is connecting to are jointly managing the tokens. A client MAY use a token from any previous connection to that server.

A token allows a server to correlate activity between the connection where the token was issued and any connection where it is used. Clients that want to break continuity of identity with a server can discard tokens provided using the NEW_TOKEN frame. In comparison, a token obtained in a Retry packet MUST be used immediately during the connection attempt and cannot be used in subsequent connection attempts.

A client SHOULD NOT reuse a NEW_TOKEN token for different connection attempts. Reusing a token allows connections to be linked by entities on the network path; see Section 9.5.

Clients might receive multiple tokens on a single connection. Aside from preventing linkability, any token can be used in any connection attempt. Servers can send additional tokens to either enable address validation for multiple connection attempts or to replace older tokens that might become invalid. For a client, this ambiguity means that sending the most recent unused token is most likely to be effective. Though saving and using older tokens has no negative consequences, clients can regard older tokens as being less likely to be useful to the server for address validation.

When a server receives an Initial packet with an address validation token, it MUST attempt to validate the token, unless it has already completed address validation. If the token is invalid then the server SHOULD proceed as if the client did not have a validated address, including potentially sending a Retry. Tokens provided with NEW_TOKEN frames and Retry packets can be distinguished by servers (see Section 8.1.1), and the latter validated more strictly. If the validation succeeds, the server SHOULD then allow the handshake to proceed.

Note: The rationale for treating the client as unvalidated rather than discarding the packet is that the client might have received the token in a previous connection using the NEW_TOKEN frame, and if the server has lost state, it might be unable to validate the token at all, leading to connection failure if the packet is discarded.

In a stateless design, a server can use encrypted and authenticated tokens to pass information to clients that the server can later recover and use to validate a client address. Tokens are not integrated into the cryptographic handshake and so they are not authenticated. For instance, a client might be able to reuse a token. To avoid attacks that exploit this property, a server can limit its use of tokens to only the information needed to validate client addresses.

Clients MAY use tokens obtained on one connection for any connection attempt using the same version. When selecting a token to use, clients do not need to consider other properties of the connection that is being attempted, including the choice of possible application protocols, session tickets, or other connection properties.

8.1.4. Address Validation Token Integrity

An address validation token MUST be difficult to guess. Including a random value with at least 128 bits of entropy in the token would be sufficient, but this depends on the server remembering the value it sends to clients.

A token-based scheme allows the server to offload any state associated with validation to the client. For this design to work, the token MUST be covered by integrity protection against modification or falsification by clients. Without integrity protection, malicious clients could generate or guess values for tokens that would be accepted by the server. Only the server requires access to the integrity protection key for tokens.

There is no need for a single well-defined format for the token because the server that generates the token also consumes it. Tokens sent in Retry packets SHOULD include information that allows the server to verify that the source IP address and port in client packets remain constant.

Tokens sent in NEW_TOKEN frames MUST include information that allows the server to verify that the client IP address has not changed from when the token was issued. Servers can use tokens from NEW_TOKEN in deciding not to send a Retry packet, even if the client address has changed. If the client IP address has changed, the server MUST adhere to the anti-amplification limit; see Section 8. Note that in the presence of NAT, this requirement might be insufficient to protect other hosts that share the NAT from amplification attack.

Attackers could replay tokens to use servers as amplifiers in DDoS attacks. To protect against such attacks, servers MUST ensure that replay of tokens is prevented or limited. Servers SHOULD ensure that tokens sent in Retry packets are only accepted for a short time, as they are returned immediately by clients. Tokens that are provided in NEW_TOKEN frames (Section 19.7) need to be valid for longer, but SHOULD NOT be accepted multiple times. Servers are encouraged to allow tokens to be used only once, if possible; tokens MAY include additional information about clients to further narrow applicability or reuse.

8.2. Path Validation

Path validation is used by both peers during connection migration (see Section 9) to verify reachability after a change of address. In path validation, endpoints test reachability between a specific local address and a specific peer address, where an address is the two-tuple of IP address and port.

Path validation tests that packets sent on a path to a peer are received by that peer. Path validation is used to ensure that packets received from a migrating peer do not carry a spoofed source address.

Path validation does not validate that a peer can send in the return direction. Acknowledgments cannot be used for return path validation because they contain insufficient entropy and might be spoofed. Endpoints independently determine reachability on each direction of a path, and therefore return reachability can only be established by the peer.

Path validation can be used at any time by either endpoint. For instance, an endpoint might check that a peer is still in possession of its address after a period of quiescence.

Path validation is not designed as a NAT traversal mechanism. Though the mechanism described here might be effective for the creation of NAT bindings that support NAT traversal, the expectation is that one or other peer is able to receive packets without first having sent a packet on that path. Effective NAT traversal needs additional synchronization mechanisms that are not provided here.

An endpoint MAY include other frames with the PATH_CHALLENGE and PATH_RESPONSE frames used for path validation. In particular, an endpoint can include PADDING frames with a PATH_CHALLENGE frame for Path Maximum Transmission Unit Discovery (PMTUD; see Section 14.2.1); it can also include its own PATH_CHALLENGE frame with a PATH_RESPONSE frame.

An endpoint uses a new connection ID for probes sent from a new local address; see Section 9.5. When probing a new path, an endpoint can ensure that its peer has an unused connection ID available for responses. Sending NEW_CONNECTION_ID and PATH_CHALLENGE frames in the same packet, if the peer's `active_connection_id_limit` permits, ensures that an unused connection ID will be available to the peer when sending a response.

An endpoint can choose to simultaneously probe multiple paths. The number of simultaneous paths used for probes is limited by the number of extra connection IDs its peer has previously supplied, since each new local address used for a probe requires a previously unused connection ID.

8.2.1. Initiating Path Validation

To initiate path validation, an endpoint sends a PATH_CHALLENGE frame containing an unpredictable payload on the path to be validated.

An endpoint MAY send multiple PATH_CHALLENGE frames to guard against packet loss. However, an endpoint SHOULD NOT send multiple PATH_CHALLENGE frames in a single packet.

An endpoint SHOULD NOT probe a new path with packets containing a PATH_CHALLENGE frame more frequently than it would send an Initial packet. This ensures that connection migration is no more load on a new path than establishing a new connection.

The endpoint MUST use unpredictable data in every PATH_CHALLENGE frame so that it can associate the peer's response with the corresponding PATH_CHALLENGE.

An endpoint MUST expand datagrams that contain a PATH_CHALLENGE frame to at least the smallest allowed maximum datagram size of 1200 bytes, unless the anti-amplification limit for the path does not permit sending a datagram of this size. Sending UDP datagrams of this size ensures that the network path from the endpoint to the peer can be used for QUIC; see Section 14.

When an endpoint is unable to expand the datagram size to 1200 bytes due to the anti-amplification limit, the path MTU will not be validated. To ensure that the path MTU is large enough, the endpoint MUST perform a second path validation by sending a PATH_CHALLENGE frame in a datagram of at least 1200 bytes. This additional validation can be performed after a PATH_RESPONSE is successfully received or when enough bytes have been received on the path that sending the larger datagram will not result in exceeding the anti-amplification limit.

Unlike other cases where datagrams are expanded, endpoints MUST NOT discard datagrams that appear to be too small when they contain PATH_CHALLENGE or PATH_RESPONSE.

8.2.2. Path Validation Responses

On receiving a PATH_CHALLENGE frame, an endpoint MUST respond by echoing the data contained in the PATH_CHALLENGE frame in a PATH_RESPONSE frame. An endpoint MUST NOT delay transmission of a packet containing a PATH_RESPONSE frame unless constrained by congestion control.

A PATH_RESPONSE frame MUST be sent on the network path where the PATH_CHALLENGE was received. This ensures that path validation by a peer only succeeds if the path is functional in both directions. This requirement MUST NOT be enforced by the endpoint that initiates path validation as that would enable an attack on migration; see Section 9.3.3.

An endpoint **MUST** expand datagrams that contain a `PATH_RESPONSE` frame to at least the smallest allowed maximum datagram size of 1200 bytes. This verifies that the path is able to carry datagrams of this size in both directions. However, an endpoint **MUST NOT** expand the datagram containing the `PATH_RESPONSE` if the resulting data exceeds the anti-amplification limit. This is expected to only occur if the received `PATH_CHALLENGE` was not sent in an expanded datagram.

An endpoint **MUST NOT** send more than one `PATH_RESPONSE` frame in response to one `PATH_CHALLENGE` frame; see Section 13.3. The peer is expected to send more `PATH_CHALLENGE` frames as necessary to evoke additional `PATH_RESPONSE` frames.

8.2.3. Successful Path Validation

Path validation succeeds when a `PATH_RESPONSE` frame is received that contains the data that was sent in a previous `PATH_CHALLENGE` frame. A `PATH_RESPONSE` frame received on any network path validates the path on which the `PATH_CHALLENGE` was sent.

If an endpoint sends a `PATH_CHALLENGE` frame in a datagram that is not expanded to at least 1200 bytes, and if the response to it validates the peer address, the path is validated but not the path MTU. As a result, the endpoint can now send more than three times the amount of data that has been received. However, the endpoint **MUST** initiate another path validation with an expanded datagram to verify that the path supports the required MTU.

Receipt of an acknowledgment for a packet containing a `PATH_CHALLENGE` frame is not adequate validation, since the acknowledgment can be spoofed by a malicious peer.

8.2.4. Failed Path Validation

Path validation only fails when the endpoint attempting to validate the path abandons its attempt to validate the path.

Endpoints **SHOULD** abandon path validation based on a timer. When setting this timer, implementations are cautioned that the new path could have a longer round-trip time than the original. A value of three times the larger of the current Probe Timeout (PTO) or the PTO for the new path (that is, using `kInitialRtt` as defined in [QUIC-RECOVERY]) is **RECOMMENDED**.

This timeout allows for multiple PTOs to expire prior to failing path validation, so that loss of a single `PATH_CHALLENGE` or `PATH_RESPONSE` frame does not cause path validation failure.

Note that the endpoint might receive packets containing other frames on the new path, but a `PATH_RESPONSE` frame with appropriate data is required for path validation to succeed.

When an endpoint abandons path validation, it determines that the path is unusable. This does not necessarily imply a failure of the connection - endpoints can continue sending packets over other paths as appropriate. If no paths are available, an endpoint can wait for a new path to become available or close the connection. An endpoint that has no valid network path to its peer MAY signal this using the `NO_VIABLE_PATH` connection error, noting that this is only possible if the network path exists but does not support the required MTU (Section 14).

A path validation might be abandoned for other reasons besides failure. Primarily, this happens if a connection migration to a new path is initiated while a path validation on the old path is in progress.

9. Connection Migration

The use of a connection ID allows connections to survive changes to endpoint addresses (IP address and port), such as those caused by an endpoint migrating to a new network. This section describes the process by which an endpoint migrates to a new address.

The design of QUIC relies on endpoints retaining a stable address for the duration of the handshake. An endpoint MUST NOT initiate connection migration before the handshake is confirmed, as defined in section 4.1.2 of [QUIC-TLS].

If the peer sent the `disable_active_migration` transport parameter, an endpoint also MUST NOT send packets (including probing packets; see Section 9.1) from a different local address to the address the peer used during the handshake, unless the endpoint has acted on a `preferred_address` transport parameter from the peer. If the peer violates this requirement, the endpoint MUST either drop the incoming packets on that path without generating a stateless reset or proceed with path validation and allow the peer to migrate. Generating a stateless reset or closing the connection would allow third parties in the network to cause connections to close by spoofing or otherwise manipulating observed traffic.

Not all changes of peer address are intentional, or active, migrations. The peer could experience NAT rebinding: a change of address due to a middlebox, usually a NAT, allocating a new outgoing port or even a new outgoing IP address for a flow. An endpoint **MUST** perform path validation (Section 8.2) if it detects any change to a peer's address, unless it has previously validated that address.

When an endpoint has no validated path on which to send packets, it **MAY** discard connection state. An endpoint capable of connection migration **MAY** wait for a new path to become available before discarding connection state.

This document limits migration of connections to new client addresses, except as described in Section 9.6. Clients are responsible for initiating all migrations. Servers do not send non-probing packets (see Section 9.1) toward a client address until they see a non-probing packet from that address. If a client receives packets from an unknown server address, the client **MUST** discard these packets.

9.1. Probing a New Path

An endpoint **MAY** probe for peer reachability from a new local address using path validation (Section 8.2) prior to migrating the connection to the new local address. Failure of path validation simply means that the new path is not usable for this connection. Failure to validate a path does not cause the connection to end unless there are no valid alternative paths available.

PATH_CHALLENGE, PATH_RESPONSE, NEW_CONNECTION_ID, and PADDING frames are "probing frames", and all other frames are "non-probing frames". A packet containing only probing frames is a "probing packet", and a packet containing any other frame is a "non-probing packet".

9.2. Initiating Connection Migration

An endpoint can migrate a connection to a new local address by sending packets containing non-probing frames from that address.

Each endpoint validates its peer's address during connection establishment. Therefore, a migrating endpoint can send to its peer knowing that the peer is willing to receive at the peer's current address. Thus an endpoint can migrate to a new local address without first validating the peer's address.

To establish reachability on the new path, an endpoint initiates path validation (Section 8.2) on the new path. An endpoint MAY defer path validation until after a peer sends the next non-probing frame to its new address.

When migrating, the new path might not support the endpoint's current sending rate. Therefore, the endpoint resets its congestion controller and RTT estimate, as described in Section 9.4.

The new path might not have the same ECN capability. Therefore, the endpoint validates ECN capability as described in Section 13.4.

9.3. Responding to Connection Migration

Receiving a packet from a new peer address containing a non-probing frame indicates that the peer has migrated to that address.

If the recipient permits the migration, it MUST send subsequent packets to the new peer address and MUST initiate path validation (Section 8.2) to verify the peer's ownership of the address if validation is not already underway.

An endpoint only changes the address to which it sends packets in response to the highest-numbered non-probing packet. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.

An endpoint MAY send data to an unvalidated peer address, but it MUST protect against potential attacks as described in Section 9.3.1 and Section 9.3.2. An endpoint MAY skip validation of a peer address if that address has been seen recently. In particular, if an endpoint returns to a previously-validated path after detecting some form of spurious migration, skipping address validation and restoring loss detection and congestion state can reduce the performance impact of the attack.

After changing the address to which it sends non-probing packets, an endpoint can abandon any path validation for other addresses.

Receiving a packet from a new peer address could be the result of a NAT rebinding at the peer.

After verifying a new client address, the server SHOULD send new address validation tokens (Section 8) to the client.

9.3.1. Peer Address Spoofing

It is possible that a peer is spoofing its source address to cause an endpoint to send excessive amounts of data to an unwilling host. If the endpoint sends significantly more data than the spoofing peer, connection migration might be used to amplify the volume of data that an attacker can generate toward a victim.

As described in Section 9.3, an endpoint is required to validate a peer's new address to confirm the peer's possession of the new address. Until a peer's address is deemed valid, an endpoint limits the amount of data it sends to that address; see Section 8. In the absence of this limit, an endpoint risks being used for a denial of service attack against an unsuspecting victim.

If an endpoint skips validation of a peer address as described above, it does not need to limit its sending rate.

9.3.2. On-Path Address Spoofing

An on-path attacker could cause a spurious connection migration by copying and forwarding a packet with a spoofed address such that it arrives before the original packet. The packet with the spoofed address will be seen to come from a migrating connection, and the original packet will be seen as a duplicate and dropped. After a spurious migration, validation of the source address will fail because the entity at the source address does not have the necessary cryptographic keys to read or respond to the `PATH_CHALLENGE` frame that is sent to it even if it wanted to.

To protect the connection from failing due to such a spurious migration, an endpoint **MUST** revert to using the last validated peer address when validation of a new peer address fails. Additionally, receipt of packets with higher packet numbers from the legitimate peer address will trigger another connection migration. This will cause the validation of the address of the spurious migration to be abandoned, thus containing migrations initiated by the attacker injecting a single packet.

If an endpoint has no state about the last validated peer address, it **MUST** close the connection silently by discarding all connection state. This results in new packets on the connection being handled generically. For instance, an endpoint **MAY** send a stateless reset in response to any further incoming packets.

9.3.3. Off-Path Packet Forwarding

An off-path attacker that can observe packets might forward copies of genuine packets to endpoints. If the copied packet arrives before the genuine packet, this will appear as a NAT rebinding. Any genuine packet will be discarded as a duplicate. If the attacker is able to continue forwarding packets, it might be able to cause migration to a path via the attacker. This places the attacker on path, giving it the ability to observe or drop all subsequent packets.

This style of attack relies on the attacker using a path that has approximately the same characteristics as the direct path between endpoints. The attack is more reliable if relatively few packets are sent or if packet loss coincides with the attempted attack.

A non-probing packet received on the original path that increases the maximum received packet number will cause the endpoint to move back to that path. Eliciting packets on this path increases the likelihood that the attack is unsuccessful. Therefore, mitigation of this attack relies on triggering the exchange of packets.

In response to an apparent migration, endpoints **MUST** validate the previously active path using a `PATH_CHALLENGE` frame. This induces the sending of new packets on that path. If the path is no longer viable, the validation attempt will time out and fail; if the path is viable, but no longer desired, the validation will succeed, but only results in probing packets being sent on the path.

An endpoint that receives a `PATH_CHALLENGE` on an active path **SHOULD** send a non-probing packet in response. If the non-probing packet arrives before any copy made by an attacker, this results in the connection being migrated back to the original path. Any subsequent migration to another path restarts this entire process.

This defense is imperfect, but this is not considered a serious problem. If the path via the attack is reliably faster than the original path despite multiple attempts to use that original path, it is not possible to distinguish between attack and an improvement in routing.

An endpoint could also use heuristics to improve detection of this style of attack. For instance, NAT rebinding is improbable if packets were recently received on the old path; similarly, rebinding is rare on IPv6 paths. Endpoints can also look for duplicated packets. Conversely, a change in connection ID is more likely to indicate an intentional migration rather than an attack.

9.4. Loss Detection and Congestion Control

The capacity available on the new path might not be the same as the old path. Packets sent on the old path **MUST NOT** contribute to congestion control or RTT estimation for the new path.

On confirming a peer's ownership of its new address, an endpoint **MUST** immediately reset the congestion controller and round-trip time estimator for the new path to initial values (see Appendices A.3 and B.3 in [QUIC-RECOVERY]) unless the only change in the peer's address is its port number. Because port-only changes are commonly the result of NAT rebinding or other middlebox activity, the endpoint **MAY** instead retain its congestion control state and round-trip estimate in those cases instead of reverting to initial values. In cases where congestion control state retained from an old path is used on a new path with substantially different characteristics, a sender could transmit too aggressively until the congestion controller and the RTT estimator have adapted. Generally, implementations are advised to be cautious when using previous values on a new path.

There could be apparent reordering at the receiver when an endpoint sends data and probes from/to multiple addresses during the migration period, since the two resulting paths could have different round-trip times. A receiver of packets on multiple paths will still send ACK frames covering all received packets.

While multiple paths might be used during connection migration, a single congestion control context and a single loss recovery context (as described in [QUIC-RECOVERY]) could be adequate. For instance, an endpoint might delay switching to a new congestion control context until it is confirmed that an old path is no longer needed (such as the case in Section 9.3.3).

A sender can make exceptions for probe packets so that their loss detection is independent and does not unduly cause the congestion controller to reduce its sending rate. An endpoint might set a separate timer when a `PATH_CHALLENGE` is sent, which is cancelled if the corresponding `PATH_RESPONSE` is received. If the timer fires before the `PATH_RESPONSE` is received, the endpoint might send a new `PATH_CHALLENGE`, and restart the timer for a longer period of time. This timer **SHOULD** be set as described in Section 6.2.1 of [QUIC-RECOVERY] and **MUST NOT** be more aggressive.

9.5. Privacy Implications of Connection Migration

Using a stable connection ID on multiple network paths would allow a passive observer to correlate activity between those paths. An endpoint that moves between networks might not wish to have their activity correlated by any entity other than their peer, so different connection IDs are used when sending from different local addresses, as discussed in Section 5.1. For this to be effective, endpoints need to ensure that connection IDs they provide cannot be linked by any other entity.

At any time, endpoints MAY change the Destination Connection ID they transmit with to a value that has not been used on another path.

An endpoint MUST NOT reuse a connection ID when sending from more than one local address, for example when initiating connection migration as described in Section 9.2 or when probing a new network path as described in Section 9.1.

Similarly, an endpoint MUST NOT reuse a connection ID when sending to more than one destination address. Due to network changes outside the control of its peer, an endpoint might receive packets from a new source address with the same destination connection ID, in which case it MAY continue to use the current connection ID with the new remote address while still sending from the same local address.

These requirements regarding connection ID reuse apply only to the sending of packets, as unintentional changes in path without a change in connection ID are possible. For example, after a period of network inactivity, NAT rebinding might cause packets to be sent on a new path when the client resumes sending. An endpoint responds to such an event as described in Section 9.3.

Using different connection IDs for packets sent in both directions on each new network path eliminates the use of the connection ID for linking packets from the same connection across different network paths. Header protection ensures that packet numbers cannot be used to correlate activity. This does not prevent other properties of packets, such as timing and size, from being used to correlate activity.

An endpoint SHOULD NOT initiate migration with a peer that has requested a zero-length connection ID, because traffic over the new path might be trivially linkable to traffic over the old one. If the server is able to associate packets with a zero-length connection ID to the right connection, it means that the server is using other information to demultiplex packets. For example, a server might provide a unique address to every client, for instance using HTTP

alternative services [ALTSVC]. Information that might allow correct routing of packets across multiple network paths will also allow activity on those paths to be linked by entities other than the peer.

A client might wish to reduce linkability by switching to a new connection ID, source UDP port, or IP address (see [RFC4941]) when sending traffic after a period of inactivity. Changing the address from which it sends packets at the same time might cause the server to detect a connection migration. This ensures that the mechanisms that support migration are exercised even for clients that do not experience NAT rebindings or genuine migrations. Changing address can cause a peer to reset its congestion control state (see Section 9.4), so addresses SHOULD only be changed infrequently.

An endpoint that exhausts available connection IDs cannot probe new paths or initiate migration, nor can it respond to probes or attempts by its peer to migrate. To ensure that migration is possible and packets sent on different paths cannot be correlated, endpoints SHOULD provide new connection IDs before peers migrate; see Section 5.1.1. If a peer might have exhausted available connection IDs, a migrating endpoint could include a NEW_CONNECTION_ID frame in all packets sent on a new network path.

9.6. Server's Preferred Address

QUIC allows servers to accept connections on one IP address and attempt to transfer these connections to a more preferred address shortly after the handshake. This is particularly useful when clients initially connect to an address shared by multiple servers but would prefer to use a unicast address to ensure connection stability. This section describes the protocol for migrating a connection to a preferred server address.

Migrating a connection to a new server address mid-connection is not supported by the version of QUIC specified in this document. If a client receives packets from a new server address when the client has not initiated a migration to that address, the client SHOULD discard these packets.

9.6.1. Communicating a Preferred Address

A server conveys a preferred address by including the preferred_address transport parameter in the TLS handshake.

Servers MAY communicate a preferred address of each address family (IPv4 and IPv6) to allow clients to pick the one most suited to their network attachment.

Once the handshake is confirmed, the client SHOULD select one of the two addresses provided by the server and initiate path validation (see Section 8.2). A client constructs packets using any previously unused active connection ID, taken from either the `preferred_address` transport parameter or a `NEW_CONNECTION_ID` frame.

As soon as path validation succeeds, the client SHOULD begin sending all future packets to the new server address using the new connection ID and discontinue use of the old server address. If path validation fails, the client MUST continue sending all future packets to the server's original IP address.

9.6.2. Migration to a Preferred Address

A client that migrates to a preferred address MUST validate the address it chooses before migrating; see Section 21.5.3.

A server might receive a packet addressed to its preferred IP address at any time after it accepts a connection. If this packet contains a `PATH_CHALLENGE` frame, the server sends a packet containing a `PATH_RESPONSE` frame as per Section 8.2. The server MUST send non-probing packets from its original address until it receives a non-probing packet from the client at its preferred address and until the server has validated the new path.

The server MUST probe on the path toward the client from its preferred address. This helps to guard against spurious migration initiated by an attacker.

Once the server has completed its path validation and has received a non-probing packet with a new largest packet number on its preferred address, the server begins sending non-probing packets to the client exclusively from its preferred IP address. The server SHOULD drop newer packets for this connection that are received on the old IP address. The server MAY continue to process delayed packets that are received on the old IP address.

The addresses that a server provides in the `preferred_address` transport parameter are only valid for the connection in which they are provided. A client MUST NOT use these for other connections, including connections that are resumed from the current connection.

9.6.3. Interaction of Client Migration and Preferred Address

A client might need to perform a connection migration before it has migrated to the server's preferred address. In this case, the client SHOULD perform path validation to both the original and preferred server address from the client's new address concurrently.

If path validation of the server's preferred address succeeds, the client **MUST** abandon validation of the original address and migrate to using the server's preferred address. If path validation of the server's preferred address fails but validation of the server's original address succeeds, the client **MAY** migrate to its new address and continue sending to the server's original address.

If packets received at the server's preferred address have a different source address than observed from the client during the handshake, the server **MUST** protect against potential attacks as described in Section 9.3.1 and Section 9.3.2. In addition to intentional simultaneous migration, this might also occur because the client's access network used a different NAT binding for the server's preferred address.

Servers **SHOULD** initiate path validation to the client's new address upon receiving a probe packet from a different address; see Section 8.

A client that migrates to a new address **SHOULD** use a preferred address from the same address family for the server.

The connection ID provided in the `preferred_address` transport parameter is not specific to the addresses that are provided. This connection ID is provided to ensure that the client has a connection ID available for migration, but the client **MAY** use this connection ID on any path.

9.7. Use of IPv6 Flow-Label and Migration

Endpoints that send data using IPv6 **SHOULD** apply an IPv6 flow label in compliance with [RFC6437], unless the local API does not allow setting IPv6 flow labels.

The flow label generation **MUST** be designed to minimize the chances of linkability with a previously used flow label, as a stable flow label would enable correlating activity on multiple paths; see Section 9.5.

[RFC6437] suggests deriving values using a pseudorandom function to generate flow labels. Including the Destination Connection ID field in addition to source and destination addresses when generating flow labels ensures that changes are synchronized with changes in other observable identifiers. A cryptographic hash function that combines these inputs with a local secret is one way this might be implemented.

10. Connection Termination

An established QUIC connection can be terminated in one of three ways:

- * idle timeout (Section 10.1)
- * immediate close (Section 10.2)
- * stateless reset (Section 10.3)

An endpoint MAY discard connection state if it does not have a validated path on which it can send packets; see Section 8.2.

10.1. Idle Timeout

If a `max_idle_timeout` is specified by either peer in its transport parameters (Section 18.2), the connection is silently closed and its state is discarded when it remains idle for longer than the minimum of both peers `max_idle_timeout` values.

Each endpoint advertises a `max_idle_timeout`, but the effective value at an endpoint is computed as the minimum of the two advertised values (or the sole advertised value, if only one endpoint advertises a nonzero value). By announcing a `max_idle_timeout`, an endpoint commits to initiating an immediate close (Section 10.2) if it abandons the connection prior to the effective value.

An endpoint restarts its idle timer when a packet from its peer is received and processed successfully. An endpoint also restarts its idle timer when sending an ack-eliciting packet if no other ack-eliciting packets have been sent since last receiving and processing a packet. Restarting this timer when sending a packet ensures that connections are not closed after new activity is initiated.

To avoid excessively small idle timeout periods, endpoints MUST increase the idle timeout period to be at least three times the current Probe Timeout (PTO). This allows for multiple PTOs to expire, and therefore multiple probes to be sent and lost, prior to idle timeout.

10.1.1. Liveness Testing

An endpoint that sends packets close to the effective timeout risks having them be discarded at the peer, since the idle timeout period might have expired at the peer before these packets arrive.

An endpoint can send a PING or another ack-eliciting frame to test the connection for liveness if the peer could time out soon, such as within a PTO; see Section 6.2 of [QUIC-RECOVERY]. This is especially useful if any available application data cannot be safely retried. Note that the application determines what data is safe to retry.

10.1.2. Deferring Idle Timeout

An endpoint might need to send ack-eliciting packets to avoid an idle timeout if it is expecting response data, but does not have or is unable to send application data.

An implementation of QUIC might provide applications with an option to defer an idle timeout. This facility could be used when the application wishes to avoid losing state that has been associated with an open connection, but does not expect to exchange application data for some time. With this option, an endpoint could send a PING frame (Section 19.2) periodically, which will cause the peer to restart its idle timeout period. Sending a packet containing a PING frame restarts the idle timeout for this endpoint also if this is the first ack-eliciting packet sent since receiving a packet. Sending a PING frame causes the peer to respond with an acknowledgment, which also restarts the idle timeout for the endpoint.

Application protocols that use QUIC SHOULD provide guidance on when deferring an idle timeout is appropriate. Unnecessary sending of PING frames could have a detrimental effect on performance.

A connection will time out if no packets are sent or received for a period longer than the time negotiated using the `max_idle_timeout` transport parameter; see Section 10. However, state in middleboxes might time out earlier than that. Though REQ-5 in [RFC4787] recommends a 2 minute timeout interval, experience shows that sending packets every 30 seconds is necessary to prevent the majority of middleboxes from losing state for UDP flows [GATEWAY].

10.2. Immediate Close

An endpoint sends a `CONNECTION_CLOSE` frame (Section 19.19) to terminate the connection immediately. A `CONNECTION_CLOSE` frame causes all streams to immediately become closed; open streams can be assumed to be implicitly reset.

After sending a `CONNECTION_CLOSE` frame, an endpoint immediately enters the closing state; see Section 10.2.1. After receiving a `CONNECTION_CLOSE` frame, endpoints enter the draining state; see Section 10.2.2.

Violations of the protocol lead to an immediate close.

An immediate close can be used after an application protocol has arranged to close a connection. This might be after the application protocol negotiates a graceful shutdown. The application protocol can exchange messages that are needed for both application endpoints to agree that the connection can be closed, after which the application requests that QUIC close the connection. When QUIC consequently closes the connection, a CONNECTION_CLOSE frame with an application-supplied error code will be used to signal closure to the peer.

The closing and draining connection states exist to ensure that connections close cleanly and that delayed or reordered packets are properly discarded. These states SHOULD persist for at least three times the current Probe Timeout (PTO) interval as defined in [QUIC-RECOVERY].

Disposing of connection state prior to exiting the closing or draining state could result in an endpoint generating a stateless reset unnecessarily when it receives a late-arriving packet. Endpoints that have some alternative means to ensure that late-arriving packets do not induce a response, such as those that are able to close the UDP socket, MAY end these states earlier to allow for faster resource recovery. Servers that retain an open socket for accepting new connections SHOULD NOT end the closing or draining states early.

Once its closing or draining state ends, an endpoint SHOULD discard all connection state. The endpoint MAY send a stateless reset in response to any further incoming packets belonging to this connection.

10.2.1. Closing Connection State

An endpoint enters the closing state after initiating an immediate close.

In the closing state, an endpoint retains only enough information to generate a packet containing a CONNECTION_CLOSE frame and to identify packets as belonging to the connection. An endpoint in the closing state sends a packet containing a CONNECTION_CLOSE frame in response to any incoming packet that it attributes to the connection.

An endpoint SHOULD limit the rate at which it generates packets in the closing state. For instance, an endpoint could wait for a progressively increasing number of received packets or amount of time before responding to received packets.

An endpoint's selected connection ID and the QUIC version are sufficient information to identify packets for a closing connection; the endpoint MAY discard all other connection state. An endpoint that is closing is not required to process any received frame. An endpoint MAY retain packet protection keys for incoming packets to allow it to read and process a CONNECTION_CLOSE frame.

An endpoint MAY drop packet protection keys when entering the closing state and send a packet containing a CONNECTION_CLOSE frame in response to any UDP datagram that is received. However, an endpoint that discards packet protection keys cannot identify and discard invalid packets. To avoid being used for an amplification attack, such endpoints MUST limit the cumulative size of packets it sends to three times the cumulative size of the packets that are received and attributed to the connection. To minimize the state that an endpoint maintains for a closing connection, endpoints MAY send the exact same packet in response to any received packet.

Note: Allowing retransmission of a closing packet is an exception to the requirement that a new packet number be used for each packet in Section 12.3. Sending new packet numbers is primarily of advantage to loss recovery and congestion control, which are not expected to be relevant for a closed connection. Retransmitting the final packet requires less state.

While in the closing state, an endpoint could receive packets from a new source address, possibly indicating a connection migration; see Section 9. An endpoint in the closing state MUST either discard packets received from an unvalidated address or limit the cumulative size of packets it sends to an unvalidated address to three times the size of packets it receives from that address.

An endpoint is not expected to handle key updates when it is closing (Section 6 of [QUIC-TLS]). A key update might prevent the endpoint from moving from the closing state to the draining state, as the endpoint will not be able to process subsequently received packets, but it otherwise has no impact.

10.2.2. Draining Connection State

The draining state is entered once an endpoint receives a CONNECTION_CLOSE frame, which indicates that its peer is closing or draining. While otherwise identical to the closing state, an endpoint in the draining state MUST NOT send any packets. Retaining packet protection keys is unnecessary once a connection is in the draining state.

An endpoint that receives a `CONNECTION_CLOSE` frame MAY send a single packet containing a `CONNECTION_CLOSE` frame before entering the draining state, using a `NO_ERROR` code if appropriate. An endpoint MUST NOT send further packets. Doing so could result in a constant exchange of `CONNECTION_CLOSE` frames until one of the endpoints exits the closing state.

An endpoint MAY enter the draining state from the closing state if it receives a `CONNECTION_CLOSE` frame, which indicates that the peer is also closing or draining. In this case, the draining state ends when the closing state would have ended. In other words, the endpoint uses the same end time, but ceases transmission of any packets on this connection.

10.2.3. Immediate Close During the Handshake

When sending `CONNECTION_CLOSE`, the goal is to ensure that the peer will process the frame. Generally, this means sending the frame in a packet with the highest level of packet protection to avoid the packet being discarded. After the handshake is confirmed (see Section 4.1.2 of [QUIC-TLS]), an endpoint MUST send any `CONNECTION_CLOSE` frames in a 1-RTT packet. However, prior to confirming the handshake, it is possible that more advanced packet protection keys are not available to the peer, so another `CONNECTION_CLOSE` frame MAY be sent in a packet that uses a lower packet protection level. More specifically:

- * A client will always know whether the server has Handshake keys (see Section 17.2.2.1), but it is possible that a server does not know whether the client has Handshake keys. Under these circumstances, a server SHOULD send a `CONNECTION_CLOSE` frame in both Handshake and Initial packets to ensure that at least one of them is processable by the client.
- * A client that sends `CONNECTION_CLOSE` in a 0-RTT packet cannot be assured that the server has accepted 0-RTT. Sending a `CONNECTION_CLOSE` frame in an Initial packet makes it more likely that the server can receive the close signal, even if the application error code might not be received.
- * Prior to confirming the handshake, a peer might be unable to process 1-RTT packets, so an endpoint SHOULD send `CONNECTION_CLOSE` in both Handshake and 1-RTT packets. A server SHOULD also send `CONNECTION_CLOSE` in an Initial packet.

Sending a `CONNECTION_CLOSE` of type `0x1d` in an Initial or Handshake packet could expose application state or be used to alter application state. A `CONNECTION_CLOSE` of type `0x1d` MUST be replaced by a

CONNECTION_CLOSE of type 0x1c when sending the frame in Initial or Handshake packets. Otherwise, information about the application state might be revealed. Endpoints MUST clear the value of the Reason Phrase field and SHOULD use the APPLICATION_ERROR code when converting to a CONNECTION_CLOSE of type 0x1c.

CONNECTION_CLOSE frames sent in multiple packet types can be coalesced into a single UDP datagram; see Section 12.2.

An endpoint can send a CONNECTION_CLOSE frame in an Initial packet. This might be in response to unauthenticated information received in Initial or Handshake packets. Such an immediate close might expose legitimate connections to a denial of service. QUIC does not include defensive measures for on-path attacks during the handshake; see Section 21.2. However, at the cost of reducing feedback about errors for legitimate peers, some forms of denial of service can be made more difficult for an attacker if endpoints discard illegal packets rather than terminating a connection with CONNECTION_CLOSE. For this reason, endpoints MAY discard packets rather than immediately close if errors are detected in packets that lack authentication.

An endpoint that has not established state, such as a server that detects an error in an Initial packet, does not enter the closing state. An endpoint that has no state for the connection does not enter a closing or draining period on sending a CONNECTION_CLOSE frame.

10.3. Stateless Reset

A stateless reset is provided as an option of last resort for an endpoint that does not have access to the state of a connection. A crash or outage might result in peers continuing to send data to an endpoint that is unable to properly continue the connection. An endpoint MAY send a stateless reset in response to receiving a packet that it cannot associate with an active connection.

A stateless reset is not appropriate for indicating errors in active connections. An endpoint that wishes to communicate a fatal connection error MUST use a CONNECTION_CLOSE frame if it is able.

To support this process, an endpoint issues a stateless reset token, which is a 16-byte value that is hard to guess. If the peer subsequently receives a stateless reset, which is a UDP datagram that ends in that stateless reset token, the peer will immediately end the connection.

A stateless reset token is specific to a connection ID. An endpoint issues a stateless reset token by including the value in the Stateless Reset Token field of a NEW_CONNECTION_ID frame. Servers can also issue a `stateless_reset_token` transport parameter during the handshake that applies to the connection ID that it selected during the handshake. These exchanges are protected by encryption, so only client and server know their value. Note that clients cannot use the `stateless_reset_token` transport parameter because their transport parameters do not have confidentiality protection.

Tokens are invalidated when their associated connection ID is retired via a RETIRE_CONNECTION_ID frame (Section 19.16).

An endpoint that receives packets that it cannot process sends a packet in the following layout (see Section 1.3):

```
Stateless Reset {  
  Fixed Bits (2) = 1,  
  Unpredictable Bits (38...),  
  Stateless Reset Token (128),  
}
```

Figure 10: Stateless Reset Packet

This design ensures that a stateless reset packet is - to the extent possible - indistinguishable from a regular packet with a short header.

A stateless reset uses an entire UDP datagram, starting with the first two bits of the packet header. The remainder of the first byte and an arbitrary number of bytes following it are set to values that SHOULD be indistinguishable from random. The last 16 bytes of the datagram contain a Stateless Reset Token.

To entities other than its intended recipient, a stateless reset will appear to be a packet with a short header. For the stateless reset to appear as a valid QUIC packet, the Unpredictable Bits field needs to include at least 38 bits of data (or 5 bytes, less the two fixed bits).

The resulting minimum size of 21 bytes does not guarantee that a stateless reset is difficult to distinguish from other packets if the recipient requires the use of a connection ID. To achieve that end, the endpoint SHOULD ensure that all packets it sends are at least 22 bytes longer than the minimum connection ID length that it requests the peer to include in its packets, adding PADDING frames as necessary. This ensures that any stateless reset sent by the peer is indistinguishable from a valid packet sent to the endpoint. An

endpoint that sends a stateless reset in response to a packet that is 43 bytes or shorter SHOULD send a stateless reset that is one byte shorter than the packet it responds to.

These values assume that the Stateless Reset Token is the same length as the minimum expansion of the packet protection AEAD. Additional unpredictable bytes are necessary if the endpoint could have negotiated a packet protection scheme with a larger minimum expansion.

An endpoint MUST NOT send a stateless reset that is three times or more larger than the packet it receives to avoid being used for amplification. Section 10.3.3 describes additional limits on stateless reset size.

Endpoints MUST discard packets that are too small to be valid QUIC packets. To give an example, with the set of AEAD functions defined in [QUIC-TLS], short header packets that are smaller than 21 bytes are never valid.

Endpoints MUST send stateless reset packets formatted as a packet with a short header. However, endpoints MUST treat any packet ending in a valid stateless reset token as a stateless reset, as other QUIC versions might allow the use of a long header.

An endpoint MAY send a stateless reset in response to a packet with a long header. Sending a stateless reset is not effective prior to the stateless reset token being available to a peer. In this QUIC version, packets with a long header are only used during connection establishment. Because the stateless reset token is not available until connection establishment is complete or near completion, ignoring an unknown packet with a long header might be as effective as sending a stateless reset.

An endpoint cannot determine the Source Connection ID from a packet with a short header, therefore it cannot set the Destination Connection ID in the stateless reset packet. The Destination Connection ID will therefore differ from the value used in previous packets. A random Destination Connection ID makes the connection ID appear to be the result of moving to a new connection ID that was provided using a NEW_CONNECTION_ID frame (Section 19.15).

Using a randomized connection ID results in two problems:

- * The packet might not reach the peer. If the Destination Connection ID is critical for routing toward the peer, then this packet could be incorrectly routed. This might also trigger another Stateless Reset in response; see Section 10.3.3. A

Stateless Reset that is not correctly routed is an ineffective error detection and recovery mechanism. In this case, endpoints will need to rely on other methods - such as timers - to detect that the connection has failed.

- * The randomly generated connection ID can be used by entities other than the peer to identify this as a potential stateless reset. An endpoint that occasionally uses different connection IDs might introduce some uncertainty about this.

This stateless reset design is specific to QUIC version 1. An endpoint that supports multiple versions of QUIC needs to generate a stateless reset that will be accepted by peers that support any version that the endpoint might support (or might have supported prior to losing state). Designers of new versions of QUIC need to be aware of this and either reuse this design, or use a portion of the packet other than the last 16 bytes for carrying data.

10.3.1. Detecting a Stateless Reset

An endpoint detects a potential stateless reset using the trailing 16 bytes of the UDP datagram. An endpoint remembers all Stateless Reset Tokens associated with the connection IDs and remote addresses for datagrams it has recently sent. This includes Stateless Reset Tokens from NEW_CONNECTION_ID frames and the server's transport parameters but excludes Stateless Reset Tokens associated with connection IDs that are either unused or retired. The endpoint identifies a received datagram as a stateless reset by comparing the last 16 bytes of the datagram with all Stateless Reset Tokens associated with the remote address on which the datagram was received.

This comparison can be performed for every inbound datagram. Endpoints MAY skip this check if any packet from a datagram is successfully processed. However, the comparison MUST be performed when the first packet in an incoming datagram either cannot be associated with a connection, or cannot be decrypted.

An endpoint MUST NOT check for any Stateless Reset Tokens associated with connection IDs it has not used or for connection IDs that have been retired.

When comparing a datagram to Stateless Reset Token values, endpoints MUST perform the comparison without leaking information about the value of the token. For example, performing this comparison in constant time protects the value of individual Stateless Reset Tokens from information leakage through timing side channels. Another approach would be to store and compare the transformed values of Stateless Reset Tokens instead of the raw token values, where the

transformation is defined as a cryptographically-secure pseudo-random function using a secret key (e.g., block cipher, HMAC [RFC2104]). An endpoint is not expected to protect information about whether a packet was successfully decrypted, or the number of valid Stateless Reset Tokens.

If the last 16 bytes of the datagram are identical in value to a Stateless Reset Token, the endpoint **MUST** enter the draining period and not send any further packets on this connection.

10.3.2. Calculating a Stateless Reset Token

The stateless reset token **MUST** be difficult to guess. In order to create a Stateless Reset Token, an endpoint could randomly generate ([RANDOM]) a secret for every connection that it creates. However, this presents a coordination problem when there are multiple instances in a cluster or a storage problem for an endpoint that might lose state. Stateless reset specifically exists to handle the case where state is lost, so this approach is suboptimal.

A single static key can be used across all connections to the same endpoint by generating the proof using a pseudorandom function that takes a static key and the connection ID chosen by the endpoint (see Section 5.1) as input. An endpoint could use HMAC [RFC2104] (for example, `HMAC(static_key, connection_id)`) or HKDF [RFC5869] (for example, using the static key as input keying material, with the connection ID as salt). The output of this function is truncated to 16 bytes to produce the Stateless Reset Token for that connection.

An endpoint that loses state can use the same method to generate a valid Stateless Reset Token. The connection ID comes from the packet that the endpoint receives.

This design relies on the peer always sending a connection ID in its packets so that the endpoint can use the connection ID from a packet to reset the connection. An endpoint that uses this design **MUST** either use the same connection ID length for all connections or encode the length of the connection ID such that it can be recovered without state. In addition, it cannot provide a zero-length connection ID.

Revealing the Stateless Reset Token allows any entity to terminate the connection, so a value can only be used once. This method for choosing the Stateless Reset Token means that the combination of connection ID and static key **MUST NOT** be used for another connection. A denial of service attack is possible if the same connection ID is used by instances that share a static key, or if an attacker can cause a packet to be routed to an instance that has no state but the

same static key; see Section 21.11. A connection ID from a connection that is reset by revealing the Stateless Reset Token MUST NOT be reused for new connections at nodes that share a static key.

The same Stateless Reset Token MUST NOT be used for multiple connection IDs. Endpoints are not required to compare new values against all previous values, but a duplicate value MAY be treated as a connection error of type `PROTOCOL_VIOLATION`.

Note that Stateless Reset packets do not have any cryptographic protection.

10.3.3. Looping

The design of a Stateless Reset is such that without knowing the stateless reset token it is indistinguishable from a valid packet. For instance, if a server sends a Stateless Reset to another server it might receive another Stateless Reset in response, which could lead to an infinite exchange.

An endpoint MUST ensure that every Stateless Reset that it sends is smaller than the packet that triggered it, unless it maintains state sufficient to prevent looping. In the event of a loop, this results in packets eventually being too small to trigger a response.

An endpoint can remember the number of Stateless Reset packets that it has sent and stop generating new Stateless Reset packets once a limit is reached. Using separate limits for different remote addresses will ensure that Stateless Reset packets can be used to close connections when other peers or connections have exhausted limits.

Reducing the size of a Stateless Reset below 41 bytes means that the packet could reveal to an observer that it is a Stateless Reset, depending upon the length of the peer's connection IDs. Conversely, refusing to send a Stateless Reset in response to a small packet might result in Stateless Reset not being useful in detecting cases of broken connections where only very small packets are sent; such failures might only be detected by other means, such as timers.

11. Error Handling

An endpoint that detects an error SHOULD signal the existence of that error to its peer. Both transport-level and application-level errors can affect an entire connection; see Section 11.1. Only application-level errors can be isolated to a single stream; see Section 11.2.

The most appropriate error code (Section 20) SHOULD be included in the frame that signals the error. Where this specification identifies error conditions, it also identifies the error code that is used; though these are worded as requirements, different implementation strategies might lead to different errors being reported. In particular, an endpoint MAY use any applicable error code when it detects an error condition; a generic error code (such as `PROTOCOL_VIOLATION` or `INTERNAL_ERROR`) can always be used in place of specific error codes.

A stateless reset (Section 10.3) is not suitable for any error that can be signaled with a `CONNECTION_CLOSE` or `RESET_STREAM` frame. A stateless reset MUST NOT be used by an endpoint that has the state necessary to send a frame on the connection.

11.1. Connection Errors

Errors that result in the connection being unusable, such as an obvious violation of protocol semantics or corruption of state that affects an entire connection, MUST be signaled using a `CONNECTION_CLOSE` frame (Section 19.19).

Application-specific protocol errors are signaled using the `CONNECTION_CLOSE` frame with a frame type of `0x1d`. Errors that are specific to the transport, including all those described in this document, are carried in the `CONNECTION_CLOSE` frame with a frame type of `0x1c`.

A `CONNECTION_CLOSE` frame could be sent in a packet that is lost. An endpoint SHOULD be prepared to retransmit a packet containing a `CONNECTION_CLOSE` frame if it receives more packets on a terminated connection. Limiting the number of retransmissions and the time over which this final packet is sent limits the effort expended on terminated connections.

An endpoint that chooses not to retransmit packets containing a `CONNECTION_CLOSE` frame risks a peer missing the first such packet. The only mechanism available to an endpoint that continues to receive data for a terminated connection is to attempt the stateless reset process (Section 10.3).

As the AEAD on Initial packets does not provide strong authentication, an endpoint MAY discard an invalid Initial packet. Discarding an Initial packet is permitted even where this specification otherwise mandates a connection error. An endpoint can only discard a packet if it does not process the frames in the packet or reverts the effects of any processing. Discarding invalid Initial packets might be used to reduce exposure to denial of service; see Section 21.2.

11.2. Stream Errors

If an application-level error affects a single stream, but otherwise leaves the connection in a recoverable state, the endpoint can send a RESET_STREAM frame (Section 19.4) with an appropriate error code to terminate just the affected stream.

Resetting a stream without the involvement of the application protocol could cause the application protocol to enter an unrecoverable state. RESET_STREAM MUST only be instigated by the application protocol that uses QUIC.

The semantics of the application error code carried in RESET_STREAM are defined by the application protocol. Only the application protocol is able to cause a stream to be terminated. A local instance of the application protocol uses a direct API call and a remote instance uses the STOP_SENDING frame, which triggers an automatic RESET_STREAM.

Application protocols SHOULD define rules for handling streams that are prematurely cancelled by either endpoint.

12. Packets and Frames

QUIC endpoints communicate by exchanging packets. Packets have confidentiality and integrity protection; see Section 12.1. Packets are carried in UDP datagrams; see Section 12.2.

This version of QUIC uses the long packet header during connection establishment; see Section 17.2. Packets with the long header are Initial (Section 17.2.2), 0-RTT (Section 17.2.3), Handshake (Section 17.2.4), and Retry (Section 17.2.5). Version negotiation uses a version-independent packet with a long header; see Section 17.2.1.

Packets with the short header are designed for minimal overhead and are used after a connection is established and 1-RTT keys are available; see Section 17.3.

12.1. Protected Packets

QUIC packets have different levels of cryptographic protection based on the type of packet. Details of packet protection are found in [QUIC-TLS]; this section includes an overview of the protections that are provided.

Version Negotiation packets have no cryptographic protection; see [QUIC-INVARIANTS].

Retry packets use an authenticated encryption with associated data function (AEAD; [AEAD]) to protect against accidental modification.

Initial packets use an AEAD, the keys for which are derived using a value that is visible on the wire. Initial packets therefore do not have effective confidentiality protection. Initial protection exists to ensure that the sender of the packet is on the network path. Any entity that receives an Initial packet from a client can recover the keys that will allow them to both read the contents of the packet and generate Initial packets that will be successfully authenticated at either endpoint. The AEAD also protects Initial packets against accidental modification.

All other packets are protected with keys derived from the cryptographic handshake. The cryptographic handshake ensures that only the communicating endpoints receive the corresponding keys for Handshake, 0-RTT, and 1-RTT packets. Packets protected with 0-RTT and 1-RTT keys have strong confidentiality and integrity protection.

The Packet Number field that appears in some packet types has alternative confidentiality protection that is applied as part of header protection; see Section 5.4 of [QUIC-TLS] for details. The underlying packet number increases with each packet sent in a given packet number space; see Section 12.3 for details.

12.2. Coalescing Packets

Initial (Section 17.2.2), 0-RTT (Section 17.2.3), and Handshake (Section 17.2.4) packets contain a Length field that determines the end of the packet. The length includes both the Packet Number and Payload fields, both of which are confidentiality protected and initially of unknown length. The length of the Payload field is learned once header protection is removed.

Using the Length field, a sender can coalesce multiple QUIC packets into one UDP datagram. This can reduce the number of UDP datagrams needed to complete the cryptographic handshake and start sending data. This can also be used to construct PMTU probes; see Section 14.4.1. Receivers **MUST** be able to process coalesced packets.

Coalescing packets in order of increasing encryption levels (Initial, 0-RTT, Handshake, 1-RTT; see Section 4.1.4 of [QUIC-TLS]) makes it more likely the receiver will be able to process all the packets in a single pass. A packet with a short header does not include a length, so it can only be the last packet included in a UDP datagram. An endpoint **SHOULD** include multiple frames in a single packet if they are to be sent at the same encryption level, instead of coalescing multiple packets at the same encryption level.

Receivers **MAY** route based on the information in the first packet contained in a UDP datagram. Senders **MUST NOT** coalesce QUIC packets with different connection IDs into a single UDP datagram. Receivers **SHOULD** ignore any subsequent packets with a different Destination Connection ID than the first packet in the datagram.

Every QUIC packet that is coalesced into a single UDP datagram is separate and complete. The receiver of coalesced QUIC packets **MUST** individually process each QUIC packet and separately acknowledge them, as if they were received as the payload of different UDP datagrams. For example, if decryption fails (because the keys are not available or any other reason), the receiver **MAY** either discard or buffer the packet for later processing and **MUST** attempt to process the remaining packets.

Retry packets (Section 17.2.5), Version Negotiation packets (Section 17.2.1), and packets with a short header (Section 17.3) do not contain a Length field and so cannot be followed by other packets in the same UDP datagram. Note also that there is no situation where a Retry or Version Negotiation packet is coalesced with another packet.

12.3. Packet Numbers

The packet number is an integer in the range 0 to $2^{62}-1$. This number is used in determining the cryptographic nonce for packet protection. Each endpoint maintains a separate packet number for sending and receiving.

Packet numbers are limited to this range because they need to be representable in whole in the Largest Acknowledged field of an ACK frame (Section 19.3). When present in a long or short header however, packet numbers are reduced and encoded in 1 to 4 bytes; see Section 17.1.

Version Negotiation (Section 17.2.1) and Retry (Section 17.2.5) packets do not include a packet number.

Packet numbers are divided into 3 spaces in QUIC:

- * Initial space: All Initial packets (Section 17.2.2) are in this space.
- * Handshake space: All Handshake packets (Section 17.2.4) are in this space.
- * Application data space: All 0-RTT (Section 17.2.3) and 1-RTT (Section 17.3.1) packets are in this space.

As described in [QUIC-TLS], each packet type uses different protection keys.

Conceptually, a packet number space is the context in which a packet can be processed and acknowledged. Initial packets can only be sent with Initial packet protection keys and acknowledged in packets that are also Initial packets. Similarly, Handshake packets are sent at the Handshake encryption level and can only be acknowledged in Handshake packets.

This enforces cryptographic separation between the data sent in the different packet number spaces. Packet numbers in each space start at packet number 0. Subsequent packets sent in the same packet number space MUST increase the packet number by at least one.

0-RTT and 1-RTT data exist in the same packet number space to make loss recovery algorithms easier to implement between the two packet types.

A QUIC endpoint MUST NOT reuse a packet number within the same packet number space in one connection. If the packet number for sending reaches $2^{62} - 1$, the sender MUST close the connection without sending a CONNECTION_CLOSE frame or any further packets; an endpoint MAY send a Stateless Reset (Section 10.3) in response to further packets that it receives.

A receiver **MUST** discard a newly unprotected packet unless it is certain that it has not processed another packet with the same packet number from the same packet number space. Duplicate suppression **MUST** happen after removing packet protection for the reasons described in Section 9.5 of [QUIC-TLS].

Endpoints that track all individual packets for the purposes of detecting duplicates are at risk of accumulating excessive state. The data required for detecting duplicates can be limited by maintaining a minimum packet number below which all packets are immediately dropped. Any minimum needs to account for large variations in round trip time, which includes the possibility that a peer might probe network paths with much larger round trip times; see Section 9.

Packet number encoding at a sender and decoding at a receiver are described in Section 17.1.

12.4. Frames and Frame Types

The payload of QUIC packets, after removing packet protection, consists of a sequence of complete frames, as shown in Figure 11. Version Negotiation, Stateless Reset, and Retry packets do not contain frames.

```
Packet Payload {  
    Frame (8..) ...,  
}
```

Figure 11: QUIC Payload

The payload of a packet that contains frames **MUST** contain at least one frame, and **MAY** contain multiple frames and multiple frame types. An endpoint **MUST** treat receipt of a packet containing no frames as a connection error of type `PROTOCOL_VIOLATION`. Frames always fit within a single QUIC packet and cannot span multiple packets.

Each frame begins with a Frame Type, indicating its type, followed by additional type-dependent fields:

```
Frame {  
    Frame Type (i),  
    Type-Dependent Fields (...),  
}
```

Figure 12: Generic Frame Layout

Table 3 lists and summarizes information about each frame type that is defined in this specification. A description of this summary is included after the table.

Type Value	Frame Type Name	Definition	Pkts	Spec
0x00	PADDING	Section 19.1	IH01	NP
0x01	PING	Section 19.2	IH01	
0x02 - 0x03	ACK	Section 19.3	IH_1	NC
0x04	RESET_STREAM	Section 19.4	__01	
0x05	STOP_SENDING	Section 19.5	__01	
0x06	CRYPTO	Section 19.6	IH_1	
0x07	NEW_TOKEN	Section 19.7	__1	
0x08 - 0x0f	STREAM	Section 19.8	__01	F
0x10	MAX_DATA	Section 19.9	__01	
0x11	MAX_STREAM_DATA	Section 19.10	__01	
0x12 - 0x13	MAX_STREAMS	Section 19.11	__01	
0x14	DATA_BLOCKED	Section 19.12	__01	
0x15	STREAM_DATA_BLOCKED	Section 19.13	__01	
0x16 - 0x17	STREAMS_BLOCKED	Section 19.14	__01	
0x18	NEW_CONNECTION_ID	Section 19.15	__01	P
0x19	RETIRE_CONNECTION_ID	Section 19.16	__01	
0x1a	PATH_CHALLENGE	Section 19.17	__01	P
0x1b	PATH_RESPONSE	Section 19.18	__1	P
0x1c - 0x1d	CONNECTION_CLOSE	Section 19.19	ih01	N
0x1e	HANDSHAKE_DONE	Section 19.20	__1	

Table 3: Frame Types

The format and semantics of each frame type are explained in more detail in Section 19. The remainder of this section provides a summary of important and general information.

The Frame Type in ACK, STREAM, MAX_STREAMS, STREAMS_BLOCKED, and CONNECTION_CLOSE frames is used to carry other frame-specific flags. For all other frames, the Frame Type field simply identifies the frame.

The "Pkts" column in Table 3 lists the types of packets that each frame type could appear in, indicated by the following characters:

I: Initial (Section 17.2.2)

H: Handshake (Section 17.2.4)

0: 0-RTT (Section 17.2.3)

1: 1-RTT (Section 17.3.1)

ih: Only a CONNECTION_CLOSE frame of type 0x1c can appear in Initial or Handshake packets.

For more detail about these restrictions, see Section 12.5. Note that all frames can appear in 1-RTT packets. An endpoint MUST treat receipt of a frame in a packet type that is not permitted as a connection error of type PROTOCOL_VIOLATION.

The "Spec" column in Table 3 summarizes any special rules governing the processing or generation of the frame type, as indicated by the following characters:

N: Packets containing only frames with this marking are not ack-eliciting; see Section 13.2.

C: Packets containing only frames with this marking do not count toward bytes in flight for congestion control purposes; see [QUIC-RECOVERY].

P: Packets containing only frames with this marking can be used to probe new network paths during connection migration; see Section 9.1.

F: The content of frames with this marking are flow controlled; see Section 4.

The "Pkts" and "Spec" columns in Table 3 do not form part of the IANA registry; see Section 22.4.

An endpoint **MUST** treat the receipt of a frame of unknown type as a connection error of type `FRAME_ENCODING_ERROR`.

All frames are idempotent in this version of QUIC. That is, a valid frame does not cause undesirable side effects or errors when received more than once.

The Frame Type field uses a variable-length integer encoding (see Section 16) with one exception. To ensure simple and efficient implementations of frame parsing, a frame type **MUST** use the shortest possible encoding. For frame types defined in this document, this means a single-byte encoding, even though it is possible to encode these values as a two-, four- or eight-byte variable-length integer. For instance, though `0x4001` is a legitimate two-byte encoding for a variable-length integer with a value of 1, PING frames are always encoded as a single byte with the value `0x01`. This rule applies to all current and future QUIC frame types. An endpoint **MAY** treat the receipt of a frame type that uses a longer encoding than necessary as a connection error of type `PROTOCOL_VIOLATION`.

12.5. Frames and Number Spaces

Some frames are prohibited in different packet number spaces. The rules here generalize those of TLS, in that frames associated with establishing the connection can usually appear in packets in any packet number space, whereas those associated with transferring data can only appear in the application data packet number space:

- * `PADDING`, `PING`, and `CRYPTO` frames **MAY** appear in any packet number space.
- * `CONNECTION_CLOSE` frames signaling errors at the QUIC layer (type `0x1c`) **MAY** appear in any packet number space. `CONNECTION_CLOSE` frames signaling application errors (type `0x1d`) **MUST** only appear in the application data packet number space.
- * `ACK` frames **MAY** appear in any packet number space, but can only acknowledge packets that appeared in that packet number space. However, as noted below, 0-RTT packets cannot contain `ACK` frames.
- * All other frame types **MUST** only be sent in the application data packet number space.

Note that it is not possible to send the following frames in 0-RTT packets for various reasons: `ACK`, `CRYPTO`, `HANDSHAKE_DONE`, `NEW_TOKEN`, `PATH_RESPONSE`, and `RETIRE_CONNECTION_ID`. A server **MAY** treat receipt of these frames in 0-RTT packets as a connection error of type `PROTOCOL_VIOLATION`.

13. Packetization and Reliability

A sender sends one or more frames in a QUIC packet; see Section 12.4.

A sender can minimize per-packet bandwidth and computational costs by including as many frames as possible in each QUIC packet. A sender MAY wait for a short period of time to collect multiple frames before sending a packet that is not maximally packed, to avoid sending out large numbers of small packets. An implementation MAY use knowledge about application sending behavior or heuristics to determine whether and for how long to wait. This waiting period is an implementation decision, and an implementation should be careful to delay conservatively, since any delay is likely to increase application-visible latency.

Stream multiplexing is achieved by interleaving STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet can include multiple STREAM frames from one or more streams.

One of the benefits of QUIC is avoidance of head-of-line blocking across multiple streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Note that when data from multiple streams is included in a single QUIC packet, loss of that packet blocks all those streams from making progress. Implementations are advised to include as few streams as necessary in outgoing packets without losing transmission efficiency to underfilled packets.

13.1. Packet Processing

A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. For STREAM frames, this means the data has been enqueued in preparation to be received by the application protocol, but it does not require that data is delivered and consumed.

Once the packet has been fully processed, a receiver acknowledges receipt by sending one or more ACK frames containing the packet number of the received packet.

An endpoint SHOULD treat receipt of an acknowledgment for a packet it did not send as a connection error of type `PROTOCOL_VIOLATION`, if it is able to detect the condition. Further discussion of how this might be achieved is in Section 21.4.

13.2. Generating Acknowledgments

Endpoints acknowledge all packets they receive and process. However, only ack-eliciting packets cause an ACK frame to be sent within the maximum ack delay. Packets that are not ack-eliciting are only acknowledged when an ACK frame is sent for other reasons.

When sending a packet for any reason, an endpoint SHOULD attempt to include an ACK frame if one has not been sent recently. Doing so helps with timely loss detection at the peer.

In general, frequent feedback from a receiver improves loss and congestion response, but this has to be balanced against excessive load generated by a receiver that sends an ACK frame in response to every ack-eliciting packet. The guidance offered below seeks to strike this balance.

13.2.1. Sending ACK Frames

Every packet SHOULD be acknowledged at least once, and ack-eliciting packets MUST be acknowledged at least once within the maximum delay an endpoint communicated using the `max_ack_delay` transport parameter; see Section 18.2. `max_ack_delay` declares an explicit contract: an endpoint promises to never intentionally delay acknowledgments of an ack-eliciting packet by more than the indicated value. If it does, any excess accrues to the RTT estimate and could result in spurious or delayed retransmissions from the peer. A sender uses the receiver's `max_ack_delay` value in determining timeouts for timer-based retransmission, as detailed in Section 6.2 of [QUIC-RECOVERY].

An endpoint MUST acknowledge all ack-eliciting Initial and Handshake packets immediately and all ack-eliciting 0-RTT and 1-RTT packets within its advertised `max_ack_delay`, with the following exception. Prior to handshake confirmation, an endpoint might not have packet protection keys for decrypting Handshake, 0-RTT, or 1-RTT packets when they are received. It might therefore buffer them and acknowledge them when the requisite keys become available.

Since packets containing only ACK frames are not congestion controlled, an endpoint MUST NOT send more than one such packet in response to receiving an ack-eliciting packet.

An endpoint MUST NOT send a non-ack-eliciting packet in response to a non-ack-eliciting packet, even if there are packet gaps that precede the received packet. This avoids an infinite feedback loop of acknowledgments, which could prevent the connection from ever becoming idle. Non-ack-eliciting packets are eventually acknowledged when the endpoint sends an ACK frame in response to other events.

In order to assist loss detection at the sender, an endpoint **SHOULD** generate and send an ACK frame without delay when it receives an ack-eliciting packet either:

- * when the received packet has a packet number less than another ack-eliciting packet that has been received, or
- * when the packet has a packet number larger than the highest-numbered ack-eliciting packet that has been received and there are missing packets between that packet and this packet.

Similarly, packets marked with the ECN Congestion Experienced (CE) codepoint in the IP header **SHOULD** be acknowledged immediately, to reduce the peer's response time to congestion events.

The algorithms in [QUIC-RECOVERY] are expected to be resilient to receivers that do not follow the guidance offered above. However, an implementation should only deviate from these requirements after careful consideration of the performance implications of a change, for connections made by the endpoint and for other users of the network.

An endpoint that is only sending ACK frames will not receive acknowledgments from its peer unless those acknowledgments are included in packets with ack-eliciting frames. An endpoint **SHOULD** send an ACK frame with other frames when there are new ack-eliciting packets to acknowledge. When only non-ack-eliciting packets need to be acknowledged, an endpoint **MAY** wait until an ack-eliciting packet has been received to include an ACK frame with outgoing frames.

A receiver **MUST NOT** send an ack-eliciting frame in all packets that would otherwise be non-ack-eliciting, to avoid an infinite feedback loop of acknowledgments.

13.2.2. Acknowledgment Frequency

A receiver determines how frequently to send acknowledgments in response to ack-eliciting packets. This determination involves a trade-off.

Endpoints rely on timely acknowledgment to detect loss; see Section 6 of [QUIC-RECOVERY]. Window-based congestion controllers, such as the one in Section 7 of [QUIC-RECOVERY], rely on acknowledgments to manage their congestion window. In both cases, delaying acknowledgments can adversely affect performance.

On the other hand, reducing the frequency of packets that carry only acknowledgments reduces packet transmission and processing cost at both endpoints. It can improve connection throughput on severely asymmetric links and reduce the volume of acknowledgment traffic using return path capacity; see Section 3 of [RFC3449].

A receiver SHOULD send an ACK frame after receiving at least two ack-eliciting packets. This recommendation is general in nature and consistent with recommendations for TCP endpoint behavior [RFC5681]. Knowledge of network conditions, knowledge of the peer's congestion controller, or further research and experimentation might suggest alternative acknowledgment strategies with better performance characteristics.

A receiver MAY process multiple available packets before determining whether to send an ACK frame in response.

13.2.3. Managing ACK Ranges

When an ACK frame is sent, one or more ranges of acknowledged packets are included. Including acknowledgments for older packets reduces the chance of spurious retransmissions caused by losing previously sent ACK frames, at the cost of larger ACK frames.

ACK frames SHOULD always acknowledge the most recently received packets, and the more out-of-order the packets are, the more important it is to send an updated ACK frame quickly, to prevent the peer from declaring a packet as lost and spuriously retransmitting the frames it contains. An ACK frame is expected to fit within a single QUIC packet. If it does not, then older ranges (those with the smallest packet numbers) are omitted.

A receiver limits the number of ACK Ranges (Section 19.3.1) it remembers and sends in ACK frames, both to limit the size of ACK frames and to avoid resource exhaustion. After receiving acknowledgments for an ACK frame, the receiver SHOULD stop tracking those acknowledged ACK Ranges. Senders can expect acknowledgments for most packets, but QUIC does not guarantee receipt of an acknowledgment for every packet that the receiver processes.

It is possible that retaining many ACK Ranges could cause an ACK frame to become too large. A receiver can discard unacknowledged ACK Ranges to limit ACK frame size, at the cost of increased retransmissions from the sender. This is necessary if an ACK frame would be too large to fit in a packet. Receivers MAY also limit ACK frame size further to preserve space for other frames or to limit the capacity that acknowledgments consume.

A receiver **MUST** retain an ACK Range unless it can ensure that it will not subsequently accept packets with numbers in that range. Maintaining a minimum packet number that increases as ranges are discarded is one way to achieve this with minimal state.

Receivers can discard all ACK Ranges, but they **MUST** retain the largest packet number that has been successfully processed as that is used to recover packet numbers from subsequent packets; see Section 17.1.

A receiver **SHOULD** include an ACK Range containing the largest received packet number in every ACK frame. The Largest Acknowledged field is used in ECN validation at a sender and including a lower value than what was included in a previous ACK frame could cause ECN to be unnecessarily disabled; see Section 13.4.2.

Section 13.2.4 describes an exemplary approach for determining what packets to acknowledge in each ACK frame. Though the goal of this algorithm is to generate an acknowledgment for every packet that is processed, it is still possible for acknowledgments to be lost.

13.2.4. Limiting Ranges by Tracking ACK Frames

When a packet containing an ACK frame is sent, the largest acknowledged in that frame can be saved. When a packet containing an ACK frame is acknowledged, the receiver can stop acknowledging packets less than or equal to the largest acknowledged in the sent ACK frame.

A receiver that sends only non-ack-eliciting packets, such as ACK frames, might not receive an acknowledgment for a long period of time. This could cause the receiver to maintain state for a large number of ACK frames for a long period of time, and ACK frames it sends could be unnecessarily large. In such a case, a receiver could send a PING or other small ack-eliciting frame occasionally, such as once per round trip, to elicit an ACK from the peer.

In cases without ACK frame loss, this algorithm allows for a minimum of 1 RTT of reordering. In cases with ACK frame loss and reordering, this approach does not guarantee that every acknowledgment is seen by the sender before it is no longer included in the ACK frame. Packets could be received out of order and all subsequent ACK frames containing them could be lost. In this case, the loss recovery algorithm could cause spurious retransmissions, but the sender will continue making forward progress.

13.2.5. Measuring and Reporting Host Delay

An endpoint measures the delays intentionally introduced between the time the packet with the largest packet number is received and the time an acknowledgment is sent. The endpoint encodes this acknowledgment delay in the ACK Delay field of an ACK frame; see Section 19.3. This allows the receiver of the ACK frame to adjust for any intentional delays, which is important for getting a better estimate of the path RTT when acknowledgments are delayed.

A packet might be held in the OS kernel or elsewhere on the host before being processed. An endpoint **MUST NOT** include delays that it does not control when populating the ACK Delay field in an ACK frame. However, endpoints **SHOULD** include buffering delays caused by unavailability of decryption keys, since these delays can be large and are likely to be non-repeating.

When the measured acknowledgment delay is larger than its `max_ack_delay`, an endpoint **SHOULD** report the measured delay. This information is especially useful during the handshake when delays might be large; see Section 13.2.1.

13.2.6. ACK Frames and Packet Protection

ACK frames **MUST** only be carried in a packet that has the same packet number space as the packet being acknowledged; see Section 12.1. For instance, packets that are protected with 1-RTT keys **MUST** be acknowledged in packets that are also protected with 1-RTT keys.

Packets that a client sends with 0-RTT packet protection **MUST** be acknowledged by the server in packets protected by 1-RTT keys. This can mean that the client is unable to use these acknowledgments if the server cryptographic handshake messages are delayed or lost. Note that the same limitation applies to other data sent by the server protected by the 1-RTT keys.

13.2.7. PADDING Frames Consume Congestion Window

Packets containing PADDING frames are considered to be in flight for congestion control purposes [QUIC-RECOVERY]. Packets containing only PADDING frames therefore consume congestion window but do not generate acknowledgments that will open the congestion window. To avoid a deadlock, a sender **SHOULD** ensure that other frames are sent periodically in addition to PADDING frames to elicit acknowledgments from the receiver.

13.3. Retransmission of Information

QUIC packets that are determined to be lost are not retransmitted whole. The same applies to the frames that are contained within lost packets. Instead, the information that might be carried in frames is sent again in new frames as needed.

New frames and packets are used to carry information that is determined to have been lost. In general, information is sent again when a packet containing that information is determined to be lost and sending ceases when a packet containing that information is acknowledged.

- * Data sent in CRYPTO frames is retransmitted according to the rules in [QUIC-RECOVERY], until all data has been acknowledged. Data in CRYPTO frames for Initial and Handshake packets is discarded when keys for the corresponding packet number space are discarded.
- * Application data sent in STREAM frames is retransmitted in new STREAM frames unless the endpoint has sent a RESET_STREAM for that stream. Once an endpoint sends a RESET_STREAM frame, no further STREAM frames are needed.
- * ACK frames carry the most recent set of acknowledgments and the acknowledgment delay from the largest acknowledged packet, as described in Section 13.2.1. Delaying the transmission of packets containing ACK frames or resending old ACK frames can cause the peer to generate an inflated RTT sample or unnecessarily disable ECN.
- * Cancellation of stream transmission, as carried in a RESET_STREAM frame, is sent until acknowledged or until all stream data is acknowledged by the peer (that is, either the "Reset Recvd" or "Data Recvd" state is reached on the sending part of the stream). The content of a RESET_STREAM frame MUST NOT change when it is sent again.
- * Similarly, a request to cancel stream transmission, as encoded in a STOP_SENDING frame, is sent until the receiving part of the stream enters either a "Data Recvd" or "Reset Recvd" state; see Section 3.5.
- * Connection close signals, including packets that contain CONNECTION_CLOSE frames, are not sent again when packet loss is detected, but as described in Section 10.

- * The current connection maximum data is sent in MAX_DATA frames. An updated value is sent in a MAX_DATA frame if the packet containing the most recently sent MAX_DATA frame is declared lost, or when the endpoint decides to update the limit. Care is necessary to avoid sending this frame too often as the limit can increase frequently and cause an unnecessarily large number of MAX_DATA frames to be sent; see Section 4.2.
- * The current maximum stream data offset is sent in MAX_STREAM_DATA frames. Like MAX_DATA, an updated value is sent when the packet containing the most recent MAX_STREAM_DATA frame for a stream is lost or when the limit is updated, with care taken to prevent the frame from being sent too often. An endpoint SHOULD stop sending MAX_STREAM_DATA frames when the receiving part of the stream enters a "Size Known" or "Reset Recvd" state.
- * The limit on streams of a given type is sent in MAX_STREAMS frames. Like MAX_DATA, an updated value is sent when a packet containing the most recent MAX_STREAMS for a stream type frame is declared lost or when the limit is updated, with care taken to prevent the frame from being sent too often.
- * Blocked signals are carried in DATA_BLOCKED, STREAM_DATA_BLOCKED, and STREAMS_BLOCKED frames. DATA_BLOCKED frames have connection scope, STREAM_DATA_BLOCKED frames have stream scope, and STREAMS_BLOCKED frames are scoped to a specific stream type. New frames are sent if packets containing the most recent frame for a scope is lost, but only while the endpoint is blocked on the corresponding limit. These frames always include the limit that is causing blocking at the time that they are transmitted.
- * A liveness or path validation check using PATH_CHALLENGE frames is sent periodically until a matching PATH_RESPONSE frame is received or until there is no remaining need for liveness or path validation checking. PATH_CHALLENGE frames include a different payload each time they are sent.
- * Responses to path validation using PATH_RESPONSE frames are sent just once. The peer is expected to send more PATH_CHALLENGE frames as necessary to evoke additional PATH_RESPONSE frames.
- * New connection IDs are sent in NEW_CONNECTION_ID frames and retransmitted if the packet containing them is lost. Retransmissions of this frame carry the same sequence number value. Likewise, retired connection IDs are sent in RETIRE_CONNECTION_ID frames and retransmitted if the packet containing them is lost.

- * NEW_TOKEN frames are retransmitted if the packet containing them is lost. No special support is made for detecting reordered and duplicated NEW_TOKEN frames other than a direct comparison of the frame contents.
- * PING and PADDING frames contain no information, so lost PING or PADDING frames do not require repair.
- * The HANDSHAKE_DONE frame MUST be retransmitted until it is acknowledged.

Endpoints SHOULD prioritize retransmission of data over sending new data, unless priorities specified by the application indicate otherwise; see Section 2.3.

Even though a sender is encouraged to assemble frames containing up-to-date information every time it sends a packet, it is not forbidden to retransmit copies of frames from lost packets. A sender that retransmits copies of frames needs to handle decreases in available payload size due to change in packet number length, connection ID length, and path MTU. A receiver MUST accept packets containing an outdated frame, such as a MAX_DATA frame carrying a smaller maximum data than one found in an older packet.

A sender SHOULD avoid retransmitting information from packets once they are acknowledged. This includes packets that are acknowledged after being declared lost, which can happen in the presence of network reordering. Doing so requires senders to retain information about packets after they are declared lost. A sender can discard this information after a period of time elapses that adequately allows for reordering, such as a PTO (Section 6.2 of [QUIC-RECOVERY]), or on other events, such as reaching a memory limit.

Upon detecting losses, a sender MUST take appropriate congestion control action. The details of loss detection and congestion control are described in [QUIC-RECOVERY].

13.4. Explicit Congestion Notification

QUIC endpoints can use Explicit Congestion Notification (ECN) [RFC3168] to detect and respond to network congestion. ECN allows an endpoint to set an ECT codepoint in the ECN field of an IP packet. A network node can then indicate congestion by setting the CE codepoint in the ECN field instead of dropping the packet [RFC8087]. Endpoints react to reported congestion by reducing their sending rate in response, as described in [QUIC-RECOVERY].

To enable ECN, a sending QUIC endpoint first determines whether a path supports ECN marking and whether the peer reports the ECN values in received IP headers; see Section 13.4.2.

13.4.1. Reporting ECN Counts

Use of ECN requires the receiving endpoint to read the ECN field from an IP packet, which is not possible on all platforms. If an endpoint does not implement ECN support or does not have access to received ECN fields, it does not report ECN counts for packets it receives.

Even if an endpoint does not set an ECT field on packets it sends, the endpoint **MUST** provide feedback about ECN markings it receives, if these are accessible. Failing to report the ECN counts will cause the sender to disable use of ECN for this connection.

On receiving an IP packet with an ECT(0), ECT(1) or CE codepoint, an ECN-enabled endpoint accesses the ECN field and increases the corresponding ECT(0), ECT(1), or CE count. These ECN counts are included in subsequent ACK frames; see Section 13.2 and Section 19.3.

Each packet number space maintains separate acknowledgment state and separate ECN counts. Coalesced QUIC packets (see Section 12.2) share the same IP header so the ECN counts are incremented once for each coalesced QUIC packet.

For example, if one each of an Initial, Handshake, and 1-RTT QUIC packet are coalesced into a single UDP datagram, the ECN counts for all three packet number spaces will be incremented by one each, based on the ECN field of the single IP header.

ECN counts are only incremented when QUIC packets from the received IP packet are processed. As such, duplicate QUIC packets are not processed and do not increase ECN counts; see Section 21.10 for relevant security concerns.

13.4.2. ECN Validation

It is possible for faulty network devices to corrupt or erroneously drop packets that carry a non-zero ECN codepoint. To ensure connectivity in the presence of such devices, an endpoint validates the ECN counts for each network path and disables use of ECN on that path if errors are detected.

To perform ECN validation for a new path:

- * The endpoint sets an ECT(0) codepoint in the IP header of early outgoing packets sent on a new path to the peer ([RFC8311]).

- * The endpoint monitors whether all packets sent with an ECT codepoint are eventually deemed lost (Section 6 of [QUIC-RECOVERY]), indicating that ECN validation has failed.

If an endpoint has cause to expect that IP packets with an ECT codepoint might be dropped by a faulty network element, the endpoint could set an ECT codepoint for only the first ten outgoing packets on a path, or for a period of three PTOs (see Section 6.2 of [QUIC-RECOVERY]). If all packets marked with non-zero ECN codepoints are subsequently lost, it can disable marking on the assumption that the marking caused the loss.

An endpoint thus attempts to use ECN and validates this for each new connection, when switching to a server's preferred address, and on active connection migration to a new path. Appendix A.4 describes one possible algorithm.

Other methods of probing paths for ECN support are possible, as are different marking strategies. Implementations MAY use other methods defined in RFCs; see [RFC8311]. Implementations that use the ECT(1) codepoint need to perform ECN validation using the reported ECT(1) counts.

13.4.2.1. Receiving ACK Frames with ECN Counts

Erroneous application of CE markings by the network can result in degraded connection performance. An endpoint that receives an ACK frame with ECN counts therefore validates the counts before using them. It performs this validation by comparing newly received counts against those from the last successfully processed ACK frame. Any increase in the ECN counts is validated based on the ECN markings that were applied to packets that are newly acknowledged in the ACK frame.

If an ACK frame newly acknowledges a packet that the endpoint sent with either the ECT(0) or ECT(1) codepoint set, ECN validation fails if the corresponding ECN counts are not present in the ACK frame. This check detects a network element that zeroes the ECN field or a peer that does not report ECN markings.

ECN validation also fails if the sum of the increase in ECT(0) and ECN-CE counts is less than the number of newly acknowledged packets that were originally sent with an ECT(0) marking. Similarly, ECN validation fails if the sum of the increases to ECT(1) and ECN-CE counts is less than the number of newly acknowledged packets sent with an ECT(1) marking. These checks can detect remarking of ECN-CE markings by the network.

An endpoint could miss acknowledgments for a packet when ACK frames are lost. It is therefore possible for the total increase in ECT(0), ECT(1), and ECN-CE counts to be greater than the number of packets that are newly acknowledged by an ACK frame. This is why ECN counts are permitted to be larger than the total number of packets that are acknowledged.

Validating ECN counts from reordered ACK frames can result in failure. An endpoint **MUST NOT** fail ECN validation as a result of processing an ACK frame that does not increase the largest acknowledged packet number.

ECN validation can fail if the received total count for either ECT(0) or ECT(1) exceeds the total number of packets sent with each corresponding ECT codepoint. In particular, validation will fail when an endpoint receives a non-zero ECN count corresponding to an ECT codepoint that it never applied. This check detects when packets are remarked to ECT(0) or ECT(1) in the network.

13.4.2.2. ECN Validation Outcomes

If validation fails, then the endpoint **MUST** disable ECN. It stops setting the ECT codepoint in IP packets that it sends, assuming that either the network path or the peer does not support ECN.

Even if validation fails, an endpoint **MAY** revalidate ECN for the same path at any later time in the connection. An endpoint could continue to periodically attempt validation.

Upon successful validation, an endpoint **MAY** continue to set an ECT codepoint in subsequent packets it sends, with the expectation that the path is ECN-capable. Network routing and path elements can however change mid-connection; an endpoint **MUST** disable ECN if validation later fails.

14. Datagram Size

A UDP datagram can include one or more QUIC packets. The datagram size refers to the total UDP payload size of a single UDP datagram carrying QUIC packets. The datagram size includes one or more QUIC packet headers and protected payloads, but not the UDP or IP headers.

The maximum datagram size is defined as the largest size of UDP payload that can be sent across a network path using a single UDP datagram. QUIC **MUST NOT** be used if the network path cannot support a maximum datagram size of at least 1200 bytes.

QUIC assumes a minimum IP packet size of at least 1280 bytes. This is the IPv6 minimum size ([IPv6]) and is also supported by most modern IPv4 networks. Assuming the minimum IP header size of 40 bytes for IPv6 and 20 bytes for IPv4 and a UDP header size of 8 bytes, this results in a maximum datagram size of 1232 bytes for IPv6 and 1252 bytes for IPv4. Thus, modern IPv4 and all IPv6 network paths are expected to be able to support QUIC.

Note: This requirement to support a UDP payload of 1200 bytes limits the space available for IPv6 extension headers to 32 bytes or IPv4 options to 52 bytes if the path only supports the IPv6 minimum MTU of 1280 bytes. This affects Initial packets and path validation.

Any maximum datagram size larger than 1200 bytes can be discovered using Path Maximum Transmission Unit Discovery (PMTUD; see Section 14.2.1) or Datagram Packetization Layer PMTU Discovery (DPLPMTUD; see Section 14.3).

Enforcement of the `max_udp_payload_size` transport parameter (Section 18.2) might act as an additional limit on the maximum datagram size. A sender can avoid exceeding this limit, once the value is known. However, prior to learning the value of the transport parameter, endpoints risk datagrams being lost if they send datagrams larger than the smallest allowed maximum datagram size of 1200 bytes.

UDP datagrams MUST NOT be fragmented at the IP layer. In IPv4 ([IPv4]), the DF bit MUST be set if possible, to prevent fragmentation on the path.

QUIC sometimes requires datagrams to be no smaller than a certain size; see Section 8.1 as an example. However, the size of a datagram is not authenticated. That is, if an endpoint receives a datagram of a certain size, it cannot know that the sender sent the datagram at the same size. Therefore, an endpoint MUST NOT close a connection when it receives a datagram that does not meet size constraints; the endpoint MAY however discard such datagrams.

14.1. Initial Datagram Size

A client MUST expand the payload of all UDP datagrams carrying Initial packets to at least the smallest allowed maximum datagram size of 1200 bytes by adding PADDING frames to the Initial packet or by coalescing the Initial packet; see Section 12.2. Initial packets can even be coalesced with invalid packets, which a receiver will discard. Similarly, a server MUST expand the payload of all UDP datagrams carrying ack-eliciting Initial packets to at least the smallest allowed maximum datagram size of 1200 bytes.

Sending UDP datagrams of this size ensures that the network path supports a reasonable Path Maximum Transmission Unit (PMTU), in both directions. Additionally, a client that expands Initial packets helps reduce the amplitude of amplification attacks caused by server responses toward an unverified client address; see Section 8.

Datagrams containing Initial packets MAY exceed 1200 bytes if the sender believes that the network path and peer both support the size that it chooses.

A server MUST discard an Initial packet that is carried in a UDP datagram with a payload that is smaller than the smallest allowed maximum datagram size of 1200 bytes. A server MAY also immediately close the connection by sending a CONNECTION_CLOSE frame with an error code of PROTOCOL_VIOLATION; see Section 10.2.3.

The server MUST also limit the number of bytes it sends before validating the address of the client; see Section 8.

14.2. Path Maximum Transmission Unit

The Path Maximum Transmission Unit (PMTU) is the maximum size of the entire IP packet including the IP header, UDP header, and UDP payload. The UDP payload includes one or more QUIC packet headers and protected payloads. The PMTU can depend on path characteristics, and can therefore change over time. The largest UDP payload an endpoint sends at any given time is referred to as the endpoint's maximum datagram size.

An endpoint SHOULD use DPLPMTUD (Section 14.3) or PMTUD (Section 14.2.1) to determine whether the path to a destination will support a desired maximum datagram size without fragmentation. In the absence of these mechanisms, QUIC endpoints SHOULD NOT send datagrams larger than the smallest allowed maximum datagram size.

Both DPLPMTUD and PMTUD send datagrams that are larger than the current maximum datagram size, referred to as PMTU probes. All QUIC packets that are not sent in a PMTU probe SHOULD be sized to fit within the maximum datagram size to avoid the datagram being fragmented or dropped ([RFC8085]).

If a QUIC endpoint determines that the PMTU between any pair of local and remote IP addresses cannot support the smallest allowed maximum datagram size of 1200 bytes, it MUST immediately cease sending QUIC packets, except for those in PMTU probes or those containing CONNECTION_CLOSE frames, on the affected path. An endpoint MAY terminate the connection if an alternative path cannot be found.

Each pair of local and remote addresses could have a different PMTU. QUIC implementations that implement any kind of PMTU discovery therefore SHOULD maintain a maximum datagram size for each combination of local and remote IP addresses.

A QUIC implementation MAY be more conservative in computing the maximum datagram size to allow for unknown tunnel overheads or IP header options/extensions.

14.2.1. Handling of ICMP Messages by PMTUD

Path Maximum Transmission Unit Discovery (PMTUD; [RFC1191], [RFC8201]) relies on reception of ICMP messages (e.g., IPv6 Packet Too Big messages) that indicate when an IP packet is dropped because it is larger than the local router MTU. DPLPMTUD can also optionally use these messages. This use of ICMP messages is potentially vulnerable to attacks by entities that cannot observe packets but might successfully guess the addresses used on the path. These attacks could reduce the PMTU to a bandwidth-inefficient value.

An endpoint MUST ignore an ICMP message that claims the PMTU has decreased below QUIC's smallest allowed maximum datagram size.

The requirements for generating ICMP ([RFC1812], [RFC4443]) state that the quoted packet should contain as much of the original packet as possible without exceeding the minimum MTU for the IP version. The size of the quoted packet can actually be smaller, or the information unintelligible, as described in Section 1.1 of [DPLPMTUD].

QUIC endpoints using PMTUD SHOULD validate ICMP messages to protect from packet injection as specified in [RFC8201] and Section 5.2 of [RFC8085]. This validation SHOULD use the quoted packet supplied in the payload of an ICMP message to associate the message with a corresponding transport connection (see Section 4.6.1 of [DPLPMTUD]). ICMP message validation MUST include matching IP addresses and UDP ports ([RFC8085]) and, when possible, connection IDs to an active QUIC session. The endpoint SHOULD ignore all ICMP messages that fail validation.

An endpoint MUST NOT increase PMTU based on ICMP messages; see Section 3, clause 6 of [DPLPMTUD]. Any reduction in QUIC's maximum datagram size in response to ICMP messages MAY be provisional until QUIC's loss detection algorithm determines that the quoted packet has actually been lost.

14.3. Datagram Packetization Layer PMTU Discovery

Datagram Packetization Layer PMTU Discovery (DPLPMTUD; [DPLPMTUD]) relies on tracking loss or acknowledgment of QUIC packets that are carried in PMTU probes. PMTU probes for DPLPMTUD that use the PADDING frame implement "Probing using padding data", as defined in Section 4.1 of [DPLPMTUD].

Endpoints SHOULD set the initial value of BASE_PLPMTU (Section 5.1 of [DPLPMTUD]) to be consistent with QUIC's smallest allowed maximum datagram size. The MIN_PLPMTU is the same as the BASE_PLPMTU.

QUIC endpoints implementing DPLPMTUD maintain a DPLPMTUD Maximum Packet Size (MPS, Section 4.4 of [DPLPMTUD]) for each combination of local and remote IP addresses. This corresponds to the maximum datagram size.

14.3.1. DPLPMTUD and Initial Connectivity

From the perspective of DPLPMTUD, QUIC is an acknowledged Packetization Layer (PL). A QUIC sender can therefore enter the DPLPMTUD BASE state (Section 5.2 of [DPLPMTUD]) when the QUIC connection handshake has been completed.

14.3.2. Validating the Network Path with DPLPMTUD

QUIC is an acknowledged PL, therefore a QUIC sender does not implement a DPLPMTUD CONFIRMATION_TIMER while in the SEARCH_COMPLETE state; see Section 5.2 of [DPLPMTUD].

14.3.3. Handling of ICMP Messages by DPLPMTUD

An endpoint using DPLPMTUD requires the validation of any received ICMP Packet Too Big (PTB) message before using the PTB information, as defined in Section 4.6 of [DPLPMTUD]. In addition to UDP port validation, QUIC validates an ICMP message by using other PL information (e.g., validation of connection IDs in the quoted packet of any received ICMP message).

The considerations for processing ICMP messages described in Section 14.2.1 also apply if these messages are used by DPLPMTUD.

14.4. Sending QUIC PMTU Probes

PMTU probes are ack-eliciting packets.

Endpoints could limit the content of PMTU probes to PING and PADDING frames, since packets that are larger than the current maximum datagram size are more likely to be dropped by the network. Loss of a QUIC packet that is carried in a PMTU probe is therefore not a reliable indication of congestion and SHOULD NOT trigger a congestion control reaction; see Section 3, Bullet 7 of [DPLPMTUD]. However, PMTU probes consume congestion window, which could delay subsequent transmission by an application.

14.4.1. PMTU Probes Containing Source Connection ID

Endpoints that rely on the destination connection ID for routing incoming QUIC packets are likely to require that the connection ID be included in PMTU probes to route any resulting ICMP messages (Section 14.2.1) back to the correct endpoint. However, only long header packets (Section 17.2) contain the Source Connection ID field, and long header packets are not decrypted or acknowledged by the peer once the handshake is complete.

One way to construct a PMTU probe is to coalesce (see Section 12.2) a packet with a long header, such as a Handshake or 0-RTT packet (Section 17.2), with a short header packet in a single UDP datagram. If the resulting PMTU probe reaches the endpoint, the packet with the long header will be ignored, but the short header packet will be acknowledged. If the PMTU probe causes an ICMP message to be sent, the first part of the probe will be quoted in that message. If the Source Connection ID field is within the quoted portion of the probe, that could be used for routing or validation of the ICMP message.

Note: The purpose of using a packet with a long header is only to ensure that the quoted packet contained in the ICMP message contains a Source Connection ID field. This packet does not need to be a valid packet and it can be sent even if there is no current use for packets of that type.

15. Versions

QUIC versions are identified using a 32-bit unsigned number.

The version 0x00000000 is reserved to represent version negotiation. This version of the specification is identified by the number 0x00000001.

Other versions of QUIC might have different properties from this version. The properties of QUIC that are guaranteed to be consistent across all versions of the protocol are described in [QUIC-INVARIANTS].

Version 0x00000001 of QUIC uses TLS as a cryptographic handshake protocol, as described in [QUIC-TLS].

Versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus documents.

Versions that follow the pattern 0x?a?a?a are reserved for use in forcing version negotiation to be exercised. That is, any version number where the low four bits of all bytes is 1010 (in binary). A client or server MAY advertise support for any of these reserved versions.

Reserved version numbers will never represent a real protocol; a client MAY use one of these version numbers with the expectation that the server will initiate version negotiation; a server MAY advertise support for one of these versions and can expect that clients ignore the value.

16. Variable-Length Integer Encoding

QUIC packets and frames commonly use a variable-length encoding for non-negative integer values. This encoding ensures that smaller integer values need fewer bytes to encode.

The QUIC variable-length integer encoding reserves the two most significant bits of the first byte to encode the base 2 logarithm of the integer encoding length in bytes. The integer value is encoded on the remaining bits, in network byte order.

This means that integers are encoded on 1, 2, 4, or 8 bytes and can encode 6-, 14-, 30-, or 62-bit values respectively. Table 4 summarizes the encoding properties.

2Bit	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 4: Summary of Integer Encodings

Examples and a sample decoding algorithm are shown in Appendix A.1.

Values do not need to be encoded on the minimum number of bytes necessary, with the sole exception of the Frame Type field; see Section 12.4.

Versions (Section 15), packet numbers sent in the header (Section 17.1), and the length of connection IDs in long header packets (Section 17.2) are described using integers, but do not use this encoding.

17. Packet Formats

All numeric values are encoded in network byte order (that is, big-endian) and all field sizes are in bits. Hexadecimal notation is used for describing the value of fields.

17.1. Packet Number Encoding and Decoding

Packet numbers are integers in the range 0 to $2^{62}-1$ (Section 12.3). When present in long or short packet headers, they are encoded in 1 to 4 bytes. The number of bits required to represent the packet number is reduced by including only the least significant bits of the packet number.

The encoded packet number is protected as described in Section 5.4 of [QUIC-TLS].

Prior to receiving an acknowledgment for a packet number space, the full packet number MUST be included; it is not to be truncated as described below.

After an acknowledgment is received for a packet number space, the sender MUST use a packet number size able to represent more than twice as large a range than the difference between the largest acknowledged packet and packet number being sent. A peer receiving the packet will then correctly decode the packet number, unless the packet is delayed in transit such that it arrives after many higher-numbered packets have been received. An endpoint SHOULD use a large enough packet number encoding to allow the packet number to be recovered even if the packet arrives after packets that are sent afterwards.

As a result, the size of the packet number encoding is at least one bit more than the base-2 logarithm of the number of contiguous unacknowledged packet numbers, including the new packet. Pseudocode and examples for packet number encoding can be found in Appendix A.2.

At a receiver, protection of the packet number is removed prior to recovering the full packet number. The full packet number is then reconstructed based on the number of significant bits present, the value of those bits, and the largest packet number received in a successfully authenticated packet. Recovering the full packet number is necessary to successfully remove packet protection.

Once header protection is removed, the packet number is decoded by finding the packet number value that is closest to the next expected packet. The next expected packet is the highest received packet number plus one. Pseudocode and an example for packet number decoding can be found in Appendix A.3.

17.2. Long Header Packets

```
Long Header Packet {  
  Header Form (1) = 1,  
  Fixed Bit (1) = 1,  
  Long Packet Type (2),  
  Type-Specific Bits (4),  
  Version (32),  
  Destination Connection ID Length (8),  
  Destination Connection ID (0..160),  
  Source Connection ID Length (8),  
  Source Connection ID (0..160),  
  Type-Specific Payload (...),  
}
```

Figure 13: Long Header Packet Format

Long headers are used for packets that are sent prior to the establishment of 1-RTT keys. Once 1-RTT keys are available, a sender switches to sending packets using the short header (Section 17.3). The long form allows for special packets – such as the Version Negotiation packet – to be represented in this uniform fixed-length packet format. Packets that use the long header contain the following fields:

Header Form: The most significant bit (0x80) of byte 0 (the first byte) is set to 1 for long headers.

Fixed Bit: The next bit (0x40) of byte 0 is set to 1, unless the packet is a Version Negotiation packet. Packets containing a zero value for this bit are not valid packets in this version and MUST be discarded. A value of 1 for this bit allows QUIC to coexist with other protocols; see [RFC7983].

Long Packet Type: The next two bits (those with a mask of 0x30) of

byte 0 contain a packet type. Packet types are listed in Table 5.

Type-Specific Bits: The semantics of the lower four bits (those with a mask of 0x0f) of byte 0 are determined by the packet type.

Version: The QUIC Version is a 32-bit field that follows the first byte. This field indicates the version of QUIC that is in use and determines how the rest of the protocol fields are interpreted.

Destination Connection ID Length: The byte following the version contains the length in bytes of the Destination Connection ID field that follows it. This length is encoded as an 8-bit unsigned integer. In QUIC version 1, this value MUST NOT exceed 20. Endpoints that receive a version 1 long header with a value larger than 20 MUST drop the packet. In order to properly form a Version Negotiation packet, servers SHOULD be able to read longer connection IDs from other QUIC versions.

Destination Connection ID: The Destination Connection ID field follows the Destination Connection ID Length field, which indicates the length of this field. Section 7.2 describes the use of this field in more detail.

Source Connection ID Length: The byte following the Destination Connection ID contains the length in bytes of the Source Connection ID field that follows it. This length is encoded as a 8-bit unsigned integer. In QUIC version 1, this value MUST NOT exceed 20 bytes. Endpoints that receive a version 1 long header with a value larger than 20 MUST drop the packet. In order to properly form a Version Negotiation packet, servers SHOULD be able to read longer connection IDs from other QUIC versions.

Source Connection ID: The Source Connection ID field follows the Source Connection ID Length field, which indicates the length of this field. Section 7.2 describes the use of this field in more detail.

Type-Specific Payload: The remainder of the packet, if any, is type-specific.

In this version of QUIC, the following packet types with the long header are defined:

Type	Name	Section
0x0	Initial	Section 17.2.2
0x1	0-RTT	Section 17.2.3
0x2	Handshake	Section 17.2.4
0x3	Retry	Section 17.2.5

Table 5: Long Header Packet Types

The header form bit, Destination and Source Connection ID lengths, Destination and Source Connection ID fields, and Version fields of a long header packet are version-independent. The other fields in the first byte are version-specific. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

The interpretation of the fields and the payload are specific to a version and packet type. While type-specific semantics for this version are described in the following sections, several long-header packets in this version of QUIC contain these additional fields:

Reserved Bits: Two bits (those with a mask of 0x0c) of byte 0 are reserved across multiple packet types. These bits are protected using header protection; see Section 5.4 of [QUIC-TLS]. The value included prior to protection MUST be set to 0. An endpoint MUST treat receipt of a packet that has a non-zero value for these bits after removing both packet and header protection as a connection error of type `PROTOCOL_VIOLATION`. Discarding such a packet after only removing header protection can expose the endpoint to attacks; see Section 9.5 of [QUIC-TLS].

Packet Number Length: In packet types that contain a Packet Number field, the least significant two bits (those with a mask of 0x03) of byte 0 contain the length of the packet number, encoded as an unsigned, two-bit integer that is one less than the length of the packet number field in bytes. That is, the length of the packet number field is the value of this field, plus one. These bits are protected using header protection; see Section 5.4 of [QUIC-TLS].

Length: The length of the remainder of the packet (that is, the Packet Number and Payload fields) in bytes, encoded as a variable-length integer (Section 16).

Packet Number: The packet number field is 1 to 4 bytes long. The

packet number is protected using header protection; see Section 5.4 of [QUIC-TLS]. The length of the packet number field is encoded in the Packet Number Length bits of byte 0; see above.

17.2.1. Version Negotiation Packet

A Version Negotiation packet is inherently not version-specific. Upon receipt by a client, it will be identified as a Version Negotiation packet based on the Version field having a value of 0.

The Version Negotiation packet is a response to a client packet that contains a version that is not supported by the server, and is only sent by servers.

The layout of a Version Negotiation packet is:

```
Version Negotiation Packet {  
  Header Form (1) = 1,  
  Unused (7),  
  Version (32) = 0,  
  Destination Connection ID Length (8),  
  Destination Connection ID (0..2040),  
  Source Connection ID Length (8),  
  Source Connection ID (0..2040),  
  Supported Version (32) ...,  
}
```

Figure 14: Version Negotiation Packet

The value in the Unused field is set to an arbitrary value by the server. Clients MUST ignore the value of this field. Where QUIC might be multiplexed with other protocols (see [RFC7983]), servers SHOULD set the most significant bit of this field (0x40) to 1 so that Version Negotiation packets appear to have the Fixed Bit field. Note that other versions of QUIC might not make a similar recommendation.

The Version field of a Version Negotiation packet MUST be set to 0x00000000.

The server MUST include the value from the Source Connection ID field of the packet it receives in the Destination Connection ID field. The value for Source Connection ID MUST be copied from the Destination Connection ID of the received packet, which is initially randomly selected by a client. Echoing both connection IDs gives clients some assurance that the server received the packet and that the Version Negotiation packet was not generated by an entity that did not observe the Initial packet.

Future versions of QUIC could have different requirements for the lengths of connection IDs. In particular, connection IDs might have a smaller minimum length or a greater maximum length. Version-specific rules for the connection ID therefore MUST NOT influence a server decision about whether to send a Version Negotiation packet.

The remainder of the Version Negotiation packet is a list of 32-bit versions that the server supports.

A Version Negotiation packet is not acknowledged. It is only sent in response to a packet that indicates an unsupported version; see Section 5.2.2.

The Version Negotiation packet does not include the Packet Number and Length fields present in other packets that use the long header form. Consequently, a Version Negotiation packet consumes an entire UDP datagram.

A server MUST NOT send more than one Version Negotiation packet in response to a single UDP datagram.

See Section 6 for a description of the version negotiation process.

17.2.2. Initial Packet

An Initial packet uses long headers with a type value of 0x0. It carries the first CRYPTO frames sent by the client and server to perform key exchange, and carries ACKs in either direction.

```
Initial Packet {  
  Header Form (1) = 1,  
  Fixed Bit (1) = 1,  
  Long Packet Type (2) = 0,  
  Reserved Bits (2),  
  Packet Number Length (2),  
  Version (32),  
  Destination Connection ID Length (8),  
  Destination Connection ID (0..160),  
  Source Connection ID Length (8),  
  Source Connection ID (0..160),  
  Token Length (i),  
  Token (..),  
  Length (i),  
  Packet Number (8..32),  
  Packet Payload (8..),  
}
```

Figure 15: Initial Packet

The Initial packet contains a long header as well as the Length and Packet Number fields; see Section 17.2. The first byte contains the Reserved and Packet Number Length bits; see also Section 17.2. Between the Source Connection ID and Length fields, there are two additional fields specific to the Initial packet.

Token Length: A variable-length integer specifying the length of the Token field, in bytes. This value is zero if no token is present. Initial packets sent by the server **MUST** set the Token Length field to zero; clients that receive an Initial packet with a non-zero Token Length field **MUST** either discard the packet or generate a connection error of type `PROTOCOL_VIOLATION`.

Token: The value of the token that was previously provided in a Retry packet or `NEW_TOKEN` frame; see Section 8.1.

Packet Payload: The payload of the packet.

In order to prevent tampering by version-unaware middleboxes, Initial packets are protected with connection- and version-specific keys (Initial keys) as described in [QUIC-TLS]. This protection does not provide confidentiality or integrity against attackers that can observe packets, but provides some level of protection against attackers that cannot observe packets.

The client and server use the Initial packet type for any packet that contains an initial cryptographic handshake message. This includes all cases where a new packet containing the initial cryptographic message needs to be created, such as the packets sent after receiving a Retry packet (Section 17.2.5).

A server sends its first Initial packet in response to a client Initial. A server **MAY** send multiple Initial packets. The cryptographic key exchange could require multiple round trips or retransmissions of this data.

The payload of an Initial packet includes a `CRYPTO` frame (or frames) containing a cryptographic handshake message, `ACK` frames, or both. `PING`, `PADDING`, and `CONNECTION_CLOSE` frames of type `0x1c` are also permitted. An endpoint that receives an Initial packet containing other frames can either discard the packet as spurious or treat it as a connection error.

The first packet sent by a client always includes a `CRYPTO` frame that contains the start or all of the first cryptographic handshake message. The first `CRYPTO` frame sent always begins at an offset of 0; see Section 7.

Note that if the server sends a TLS HelloRetryRequest (see Section 4.7 of [QUIC-TLS]), the client will send another series of Initial packets. These Initial packets will continue the cryptographic handshake and will contain CRYPTO frames starting at an offset matching the size of the CRYPTO frames sent in the first flight of Initial packets.

17.2.2.1. Abandoning Initial Packets

A client stops both sending and processing Initial packets when it sends its first Handshake packet. A server stops sending and processing Initial packets when it receives its first Handshake packet. Though packets might still be in flight or awaiting acknowledgment, no further Initial packets need to be exchanged beyond this point. Initial packet protection keys are discarded (see Section 4.9.1 of [QUIC-TLS]) along with any loss recovery and congestion control state; see Section 6.4 of [QUIC-RECOVERY].

Any data in CRYPTO frames is discarded - and no longer retransmitted - when Initial keys are discarded.

17.2.3. 0-RTT

A 0-RTT packet uses long headers with a type value of 0x1, followed by the Length and Packet Number fields; see Section 17.2. The first byte contains the Reserved and Packet Number Length bits; see Section 17.2. A 0-RTT packet is used to carry "early" data from the client to the server as part of the first flight, prior to handshake completion. As part of the TLS handshake, the server can accept or reject this early data.

See Section 2.3 of [TLS13] for a discussion of 0-RTT data and its limitations.

```
0-RTT Packet {  
  Header Form (1) = 1,  
  Fixed Bit (1) = 1,  
  Long Packet Type (2) = 1,  
  Reserved Bits (2),  
  Packet Number Length (2),  
  Version (32),  
  Destination Connection ID Length (8),  
  Destination Connection ID (0..160),  
  Source Connection ID Length (8),  
  Source Connection ID (0..160),  
  Length (i),  
  Packet Number (8..32),  
  Packet Payload (8..),  
}
```

Figure 16: 0-RTT Packet

Packet numbers for 0-RTT protected packets use the same space as 1-RTT protected packets.

After a client receives a Retry packet, 0-RTT packets are likely to have been lost or discarded by the server. A client **SHOULD** attempt to resend data in 0-RTT packets after it sends a new Initial packet. New packet numbers **MUST** be used for any new packets that are sent; as described in Section 17.2.5.3, reusing packet numbers could compromise packet protection.

A client only receives acknowledgments for its 0-RTT packets once the handshake is complete, as defined in Section 4.1.1 of [QUIC-TLS].

A client **MUST NOT** send 0-RTT packets once it starts processing 1-RTT packets from the server. This means that 0-RTT packets cannot contain any response to frames from 1-RTT packets. For instance, a client cannot send an ACK frame in a 0-RTT packet, because that can only acknowledge a 1-RTT packet. An acknowledgment for a 1-RTT packet **MUST** be carried in a 1-RTT packet.

A server **SHOULD** treat a violation of remembered limits (Section 7.4.1) as a connection error of an appropriate type (for instance, a `FLOW_CONTROL_ERROR` for exceeding stream data limits).

17.2.4. Handshake Packet

A Handshake packet uses long headers with a type value of 0x2, followed by the Length and Packet Number fields; see Section 17.2. The first byte contains the Reserved and Packet Number Length bits; see Section 17.2. It is used to carry cryptographic handshake messages and acknowledgments from the server and client.

```
Handshake Packet {  
  Header Form (1) = 1,  
  Fixed Bit (1) = 1,  
  Long Packet Type (2) = 2,  
  Reserved Bits (2),  
  Packet Number Length (2),  
  Version (32),  
  Destination Connection ID Length (8),  
  Destination Connection ID (0..160),  
  Source Connection ID Length (8),  
  Source Connection ID (0..160),  
  Length (i),  
  Packet Number (8..32),  
  Packet Payload (8..),  
}
```

Figure 17: Handshake Protected Packet

Once a client has received a Handshake packet from a server, it uses Handshake packets to send subsequent cryptographic handshake messages and acknowledgments to the server.

The Destination Connection ID field in a Handshake packet contains a connection ID that is chosen by the recipient of the packet; the Source Connection ID includes the connection ID that the sender of the packet wishes to use; see Section 7.2.

Handshake packets have their own packet number space, and thus the first Handshake packet sent by a server contains a packet number of 0.

The payload of this packet contains CRYPTO frames and could contain PING, PADDING, or ACK frames. Handshake packets MAY contain CONNECTION_CLOSE frames of type 0x1c. Endpoints MUST treat receipt of Handshake packets with other frames as a connection error of type PROTOCOL_VIOLATION.

Like Initial packets (see Section 17.2.2.1), data in CRYPTO frames for Handshake packets is discarded – and no longer retransmitted – when Handshake protection keys are discarded.

17.2.5. Retry Packet

A Retry packet uses a long packet header with a type value of 0x3. It carries an address validation token created by the server. It is used by a server that wishes to perform a retry; see Section 8.1.

```
Retry Packet {  
  Header Form (1) = 1,  
  Fixed Bit (1) = 1,  
  Long Packet Type (2) = 3,  
  Unused (4),  
  Version (32),  
  Destination Connection ID Length (8),  
  Destination Connection ID (0..160),  
  Source Connection ID Length (8),  
  Source Connection ID (0..160),  
  Retry Token (..),  
  Retry Integrity Tag (128),  
}
```

Figure 18: Retry Packet

A Retry packet (shown in Figure 18) does not contain any protected fields. The value in the Unused field is set to an arbitrary value by the server; a client MUST ignore these bits. In addition to the fields from the long header, it contains these additional fields:

Retry Token: An opaque token that the server can use to validate the client's address.

Retry Integrity Tag: See the Retry Packet Integrity section of [QUIC-TLS].

17.2.5.1. Sending a Retry Packet

The server populates the Destination Connection ID with the connection ID that the client included in the Source Connection ID of the Initial packet.

The server includes a connection ID of its choice in the Source Connection ID field. This value MUST NOT be equal to the Destination Connection ID field of the packet sent by the client. A client MUST discard a Retry packet that contains a Source Connection ID field that is identical to the Destination Connection ID field of its Initial packet. The client MUST use the value from the Source Connection ID field of the Retry packet in the Destination Connection ID field of subsequent packets that it sends.

A server MAY send Retry packets in response to Initial and 0-RTT packets. A server can either discard or buffer 0-RTT packets that it receives. A server can send multiple Retry packets as it receives Initial or 0-RTT packets. A server MUST NOT send more than one Retry packet in response to a single UDP datagram.

17.2.5.2. Handling a Retry Packet

A client MUST accept and process at most one Retry packet for each connection attempt. After the client has received and processed an Initial or Retry packet from the server, it MUST discard any subsequent Retry packets that it receives.

Clients MUST discard Retry packets that have a Retry Integrity Tag that cannot be validated; see the Retry Packet Integrity section of [QUIC-TLS]. This diminishes an attacker's ability to inject a Retry packet and protects against accidental corruption of Retry packets. A client MUST discard a Retry packet with a zero-length Retry Token field.

The client responds to a Retry packet with an Initial packet that includes the provided Retry Token to continue connection establishment.

A client sets the Destination Connection ID field of this Initial packet to the value from the Source Connection ID in the Retry packet. Changing Destination Connection ID also results in a change to the keys used to protect the Initial packet. It also sets the Token field to the token provided in the Retry. The client MUST NOT change the Source Connection ID because the server could include the connection ID as part of its token validation logic; see Section 8.1.4.

A Retry packet does not include a packet number and cannot be explicitly acknowledged by a client.

17.2.5.3. Continuing a Handshake After Retry

Subsequent Initial packets from the client include the connection ID and token values from the Retry packet. The client copies the Source Connection ID field from the Retry packet to the Destination Connection ID field and uses this value until an Initial packet with an updated value is received; see Section 7.2. The value of the Token field is copied to all subsequent Initial packets; see Section 8.1.2.

Other than updating the Destination Connection ID and Token fields, the Initial packet sent by the client is subject to the same restrictions as the first Initial packet. A client **MUST** use the same cryptographic handshake message it included in this packet. A server **MAY** treat a packet that contains a different cryptographic handshake message as a connection error or discard it. Note that including a Token field reduces the available space for the cryptographic handshake message, which might result in the client needing to send multiple Initial packets.

A client **MAY** attempt 0-RTT after receiving a Retry packet by sending 0-RTT packets to the connection ID provided by the server.

A client **MUST NOT** reset the packet number for any packet number space after processing a Retry packet. In particular, 0-RTT packets contain confidential information that will most likely be retransmitted on receiving a Retry packet. The keys used to protect these new 0-RTT packets will not change as a result of responding to a Retry packet. However, the data sent in these packets could be different than what was sent earlier. Sending these new packets with the same packet number is likely to compromise the packet protection for those packets because the same key and nonce could be used to protect different content. A server **MAY** abort the connection if it detects that the client reset the packet number.

The connection IDs used on Initial and Retry packets exchanged between client and server are copied to the transport parameters and validated as described in Section 7.3.

17.3. Short Header Packets

This version of QUIC defines a single packet type that uses the short packet header.

17.3.1. 1-RTT Packet

A 1-RTT packet uses a short packet header. It is used after the version and 1-RTT keys are negotiated.

```
1-RTT Packet {  
  Header Form (1) = 0,  
  Fixed Bit (1) = 1,  
  Spin Bit (1),  
  Reserved Bits (2),  
  Key Phase (1),  
  Packet Number Length (2),  
  Destination Connection ID (0..160),  
  Packet Number (8..32),  
  Packet Payload (8..),  
}
```

Figure 19: 1-RTT Packet

1-RTT packets contain the following fields:

Header Form: The most significant bit (0x80) of byte 0 is set to 0 for the short header.

Fixed Bit: The next bit (0x40) of byte 0 is set to 1. Packets containing a zero value for this bit are not valid packets in this version and MUST be discarded. A value of 1 for this bit allows QUIC to coexist with other protocols; see [RFC7983].

Spin Bit: The third most significant bit (0x20) of byte 0 is the latency spin bit, set as described in Section 17.4.

Reserved Bits: The next two bits (those with a mask of 0x18) of byte 0 are reserved. These bits are protected using header protection; see Section 5.4 of [QUIC-TLS]. The value included prior to protection MUST be set to 0. An endpoint MUST treat receipt of a packet that has a non-zero value for these bits, after removing both packet and header protection, as a connection error of type `PROTOCOL_VIOLATION`. Discarding such a packet after only removing header protection can expose the endpoint to attacks; see Section 9.5 of [QUIC-TLS].

Key Phase: The next bit (0x04) of byte 0 indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet. See [QUIC-TLS] for details. This bit is protected using header protection; see Section 5.4 of [QUIC-TLS].

Packet Number Length: The least significant two bits (those with a

mask of 0x03) of byte 0 contain the length of the packet number, encoded as an unsigned, two-bit integer that is one less than the length of the packet number field in bytes. That is, the length of the packet number field is the value of this field, plus one. These bits are protected using header protection; see Section 5.4 of [QUIC-TLS].

Destination Connection ID: The Destination Connection ID is a connection ID that is chosen by the intended recipient of the packet. See Section 5.1 for more details.

Packet Number: The packet number field is 1 to 4 bytes long. The packet number is protected using header protection; see Section 5.4 of [QUIC-TLS]. The length of the packet number field is encoded in Packet Number Length field. See Section 17.1 for details.

Packet Payload: 1-RTT packets always include a 1-RTT protected payload.

The header form bit and the connection ID field of a short header packet are version-independent. The remaining fields are specific to the selected QUIC version. See [QUIC-INVARIANTS] for details on how packets from different versions of QUIC are interpreted.

17.4. Latency Spin Bit

The latency spin bit, which is defined for 1-RTT packets (Section 17.3.1), enables passive latency monitoring from observation points on the network path throughout the duration of a connection. The server reflects the spin value received, while the client 'spins' it after one RTT. On-path observers can measure the time between two spin bit toggle events to estimate the end-to-end RTT of a connection.

The spin bit is only present in 1-RTT packets, since it is possible to measure the initial RTT of a connection by observing the handshake. Therefore, the spin bit is available after version negotiation and connection establishment are completed. On-path measurement and use of the latency spin bit is further discussed in [QUIC-MANAGEABILITY].

The spin bit is an OPTIONAL feature of this version of QUIC. An endpoint that does not support this feature MUST disable it, as defined below.

Each endpoint unilaterally decides if the spin bit is enabled or disabled for a connection. Implementations **MUST** allow administrators of clients and servers to disable the spin bit either globally or on a per-connection basis. Even when the spin bit is not disabled by the administrator, endpoints **MUST** disable their use of the spin bit for a random selection of at least one in every 16 network paths, or for one in every 16 connection IDs, in order to ensure that QUIC connections that disable the spin bit are commonly observed on the network. As each endpoint disables the spin bit independently, this ensures that the spin bit signal is disabled on approximately one in eight network paths.

When the spin bit is disabled, endpoints **MAY** set the spin bit to any value, and **MUST** ignore any incoming value. It is **RECOMMENDED** that endpoints set the spin bit to a random value either chosen independently for each packet or chosen independently for each connection ID.

If the spin bit is enabled for the connection, the endpoint maintains a spin value for each network path and sets the spin bit in the packet header to the currently stored value when a 1-RTT packet is sent on that path. The spin value is initialized to 0 in the endpoint for each network path. Each endpoint also remembers the highest packet number seen from its peer on each path.

When a server receives a 1-RTT packet that increases the highest packet number seen by the server from the client on a given network path, it sets the spin value for that path to be equal to the spin bit in the received packet.

When a client receives a 1-RTT packet that increases the highest packet number seen by the client from the server on a given network path, it sets the spin value for that path to the inverse of the spin bit in the received packet.

An endpoint resets the spin value for a network path to zero when changing the connection ID being used on that network path.

18. Transport Parameter Encoding

The `extension_data` field of the `quic_transport_parameters` extension defined in [QUIC-TLS] contains the QUIC transport parameters. They are encoded as a sequence of transport parameters, as shown in Figure 20:

```
Transport Parameters {  
    Transport Parameter (..) ...,  
}
```

Figure 20: Sequence of Transport Parameters

Each transport parameter is encoded as an (identifier, length, value) tuple, as shown in Figure 21:

```
Transport Parameter {  
  Transport Parameter ID (i),  
  Transport Parameter Length (i),  
  Transport Parameter Value (..),  
}
```

Figure 21: Transport Parameter Encoding

The Transport Parameter Length field contains the length of the Transport Parameter Value field in bytes.

QUIC encodes transport parameters into a sequence of bytes, which is then included in the cryptographic handshake.

18.1. Reserved Transport Parameters

Transport parameters with an identifier of the form $31 * N + 27$ for integer values of N are reserved to exercise the requirement that unknown transport parameters be ignored. These transport parameters have no semantics, and can carry arbitrary values.

18.2. Transport Parameter Definitions

This section details the transport parameters defined in this document.

Many transport parameters listed here have integer values. Those transport parameters that are identified as integers use a variable-length integer encoding; see Section 16. Transport parameters have a default value of 0 if the transport parameter is absent unless otherwise stated.

The following transport parameters are defined:

original_destination_connection_id (0x00): The value of the Destination Connection ID field from the first Initial packet sent by the client; see Section 7.3. This transport parameter is only sent by a server.

max_idle_timeout (0x01): The max idle timeout is a value in milliseconds that is encoded as an integer; see (Section 10.1). Idle timeout is disabled when both endpoints omit this transport parameter or specify a value of 0.

`stateless_reset_token (0x02)`: A stateless reset token is used in verifying a stateless reset; see Section 10.3. This parameter is a sequence of 16 bytes. This transport parameter **MUST NOT** be sent by a client, but **MAY** be sent by a server. A server that does not send this transport parameter cannot use stateless reset (Section 10.3) for the connection ID negotiated during the handshake.

`max_udp_payload_size (0x03)`: The maximum UDP payload size parameter is an integer value that limits the size of UDP payloads that the endpoint is willing to receive. UDP datagrams with payloads larger than this limit are not likely to be processed by the receiver.

The default for this parameter is the maximum permitted UDP payload of 65527. Values below 1200 are invalid.

This limit does act as an additional constraint on datagram size in the same way as the path MTU, but it is a property of the endpoint and not the path; see Section 14. It is expected that this is the space an endpoint dedicates to holding incoming packets.

`initial_max_data (0x04)`: The initial maximum data parameter is an integer value that contains the initial value for the maximum amount of data that can be sent on the connection. This is equivalent to sending a `MAX_DATA` (Section 19.9) for the connection immediately after completing the handshake.

`initial_max_stream_data_bidi_local (0x05)`: This parameter is an integer value specifying the initial flow control limit for locally-initiated bidirectional streams. This limit applies to newly created bidirectional streams opened by the endpoint that sends the transport parameter. In client transport parameters, this applies to streams with an identifier with the least significant two bits set to 0x0; in server transport parameters, this applies to streams with the least significant two bits set to 0x1.

`initial_max_stream_data_bidi_remote (0x06)`: This parameter is an integer value specifying the initial flow control limit for peer-initiated bidirectional streams. This limit applies to newly created bidirectional streams opened by the endpoint that receives the transport parameter. In client transport parameters, this applies to streams with an identifier with the least significant two bits set to 0x1; in server transport parameters, this applies to streams with the least significant two bits set to 0x0.

`initial_max_stream_data_uni (0x07)`: This parameter is an integer value specifying the initial flow control limit for unidirectional streams. This limit applies to newly created unidirectional streams opened by the endpoint that receives the transport parameter. In client transport parameters, this applies to streams with an identifier with the least significant two bits set to 0x3; in server transport parameters, this applies to streams with the least significant two bits set to 0x2.

`initial_max_streams_bidi (0x08)`: The initial maximum bidirectional streams parameter is an integer value that contains the initial maximum number of bidirectional streams the endpoint that receives this transport parameter is permitted to initiate. If this parameter is absent or zero, the peer cannot open bidirectional streams until a `MAX_STREAMS` frame is sent. Setting this parameter is equivalent to sending a `MAX_STREAMS` (Section 19.11) of the corresponding type with the same value.

`initial_max_streams_uni (0x09)`: The initial maximum unidirectional streams parameter is an integer value that contains the initial maximum number of unidirectional streams the endpoint that receives this transport parameter is permitted to initiate. If this parameter is absent or zero, the peer cannot open unidirectional streams until a `MAX_STREAMS` frame is sent. Setting this parameter is equivalent to sending a `MAX_STREAMS` (Section 19.11) of the corresponding type with the same value.

`ack_delay_exponent (0x0a)`: The acknowledgment delay exponent is an integer value indicating an exponent used to decode the ACK Delay field in the ACK frame (Section 19.3). If this value is absent, a default value of 3 is assumed (indicating a multiplier of 8). Values above 20 are invalid.

`max_ack_delay (0x0b)`: The maximum acknowledgment delay is an integer value indicating the maximum amount of time in milliseconds by which the endpoint will delay sending acknowledgments. This value SHOULD include the receiver's expected delays in alarms firing. For example, if a receiver sets a timer for 5ms and alarms commonly fire up to 1ms late, then it should send a `max_ack_delay` of 6ms. If this value is absent, a default of 25 milliseconds is assumed. Values of 2^{14} or greater are invalid.

`disable_active_migration (0x0c)`: The disable active migration transport parameter is included if the endpoint does not support active connection migration (Section 9) on the address being used during the handshake. An endpoint that receives this transport parameter MUST NOT use a new local address when sending to the address that the peer used during the handshake. This transport

parameter does not prohibit connection migration after a client has acted on a preferred_address transport parameter. This parameter is a zero-length value.

preferred_address (0x0d): The server's preferred address is used to effect a change in server address at the end of the handshake, as described in Section 9.6. This transport parameter is only sent by a server. Servers MAY choose to only send a preferred address of one address family by sending an all-zero address and port (0.0.0.0:0 or [::]:0) for the other family. IP addresses are encoded in network byte order.

The preferred_address transport parameter contains an address and port for both IP version 4 and 6. The four-byte IPv4 Address field is followed by the associated two-byte IPv4 Port field. This is followed by a 16-byte IPv6 Address field and two-byte IPv6 Port field. After address and port pairs, a Connection ID Length field describes the length of the following Connection ID field. Finally, a 16-byte Stateless Reset Token field includes the stateless reset token associated with the connection ID. The format of this transport parameter is shown in Figure 22.

The Connection ID field and the Stateless Reset Token field contain an alternative connection ID that has a sequence number of 1; see Section 5.1.1. Having these values sent alongside the preferred address ensures that there will be at least one unused active connection ID when the client initiates migration to the preferred address.

The Connection ID and Stateless Reset Token fields of a preferred address are identical in syntax and semantics to the corresponding fields of a NEW_CONNECTION_ID frame (Section 19.15). A server that chooses a zero-length connection ID MUST NOT provide a preferred address. Similarly, a server MUST NOT include a zero-length connection ID in this transport parameter. A client MUST treat violation of these requirements as a connection error of type `TRANSPORT_PARAMETER_ERROR`.

```
Preferred Address {  
  IPv4 Address (32),  
  IPv4 Port (16),  
  IPv6 Address (128),  
  IPv6 Port (16),  
  Connection ID Length (8),  
  Connection ID (...),  
  Stateless Reset Token (128),  
}
```

Figure 22: Preferred Address format

`active_connection_id_limit` (0x0e): The active connection ID limit is an integer value specifying the maximum number of connection IDs from the peer that an endpoint is willing to store. This value includes the connection ID received during the handshake, that received in the `preferred_address` transport parameter, and those received in `NEW_CONNECTION_ID` frames. The value of the `active_connection_id_limit` parameter **MUST** be at least 2. An endpoint that receives a value less than 2 **MUST** close the connection with an error of type `TRANSPORT_PARAMETER_ERROR`. If this transport parameter is absent, a default of 2 is assumed. If an endpoint issues a zero-length connection ID, it will never send a `NEW_CONNECTION_ID` frame and therefore ignores the `active_connection_id_limit` value received from its peer.

`initial_source_connection_id` (0x0f): The value that the endpoint included in the Source Connection ID field of the first Initial packet it sends for the connection; see Section 7.3.

`retry_source_connection_id` (0x10): The value that the server included in the Source Connection ID field of a Retry packet; see Section 7.3. This transport parameter is only sent by a server.

If present, transport parameters that set initial per-stream flow control limits (`initial_max_stream_data_bidi_local`, `initial_max_stream_data_bidi_remote`, and `initial_max_stream_data_uni`) are equivalent to sending a `MAX_STREAM_DATA` frame (Section 19.10) on every stream of the corresponding type immediately after opening. If the transport parameter is absent, streams of that type start with a flow control limit of 0.

A client **MUST NOT** include any server-only transport parameter: `original_destination_connection_id`, `preferred_address`, `retry_source_connection_id`, or `stateless_reset_token`. A server **MUST** treat receipt of any of these transport parameters as a connection error of type `TRANSPORT_PARAMETER_ERROR`.

19. Frame Types and Formats

As described in Section 12.4, packets contain one or more frames. This section describes the format and semantics of the core QUIC frame types.

19.1. PADDING Frames

A PADDING frame (type=0x00) has no semantic value. PADDING frames can be used to increase the size of a packet. Padding can be used to increase an initial client packet to the minimum required size, or to provide protection against traffic analysis for protected packets.

PADDING frames are formatted as shown in Figure 23, which shows that PADDING frames have no content. That is, a PADDING frame consists of the single byte that identifies the frame as a PADDING frame.

```
PADDING Frame {  
    Type (i) = 0x00,  
}
```

Figure 23: PADDING Frame Format

19.2. PING Frames

Endpoints can use PING frames (type=0x01) to verify that their peers are still alive or to check reachability to the peer.

PING frames are formatted as shown in Figure 24, which shows that PING frames have no content.

```
PING Frame {  
    Type (i) = 0x01,  
}
```

Figure 24: PING Frame Format

The receiver of a PING frame simply needs to acknowledge the packet containing this frame.

The PING frame can be used to keep a connection alive when an application or application protocol wishes to prevent the connection from timing out; see Section 10.1.2.

19.3. ACK Frames

Receivers send ACK frames (types 0x02 and 0x03) to inform senders of packets they have received and processed. The ACK frame contains one or more ACK Ranges. ACK Ranges identify acknowledged packets. If the frame type is 0x03, ACK frames also contain the cumulative count of QUIC packets with associated ECN marks received on the connection up until this point. QUIC implementations **MUST** properly handle both types and, if they have enabled ECN for packets they send, they **SHOULD** use the information in the ECN section to manage their congestion state.

QUIC acknowledgments are irrevocable. Once acknowledged, a packet remains acknowledged, even if it does not appear in a future ACK frame. This is unlike reneging for TCP SACKs ([RFC2018]).

Packets from different packet number spaces can be identified using the same numeric value. An acknowledgment for a packet needs to indicate both a packet number and a packet number space. This is accomplished by having each ACK frame only acknowledge packet numbers in the same space as the packet in which the ACK frame is contained.

Version Negotiation and Retry packets cannot be acknowledged because they do not contain a packet number. Rather than relying on ACK frames, these packets are implicitly acknowledged by the next Initial packet sent by the client.

ACK frames are formatted as shown in Figure 25.

```
ACK Frame {  
    Type (i) = 0x02..0x03,  
    Largest Acknowledged (i),  
    ACK Delay (i),  
    ACK Range Count (i),  
    First ACK Range (i),  
    ACK Range (...) ...,  
    [ECN Counts (...)],  
}
```

Figure 25: ACK Frame Format

ACK frames contain the following fields:

Largest Acknowledged: A variable-length integer representing the largest packet number the peer is acknowledging; this is usually the largest packet number that the peer has received prior to generating the ACK frame. Unlike the packet number in the QUIC long or short header, the value in an ACK frame is not truncated.

ACK Delay: A variable-length integer encoding the acknowledgment delay in microseconds; see Section 13.2.5. It is decoded by multiplying the value in the field by 2 to the power of the `ack_delay_exponent` transport parameter sent by the sender of the ACK frame; see Section 18.2. Compared to simply expressing the delay as an integer, this encoding allows for a larger range of values within the same number of bytes, at the cost of lower resolution.

ACK Range Count: A variable-length integer specifying the number of ACK Range fields in the frame.

First ACK Range: A variable-length integer indicating the number of contiguous packets preceding the Largest Acknowledged that are being acknowledged. That is, the smallest packet acknowledged in the range is determined by subtracting the First ACK Range value from the Largest Acknowledged.

ACK Ranges: Contains additional ranges of packets that are alternately not acknowledged (Gap) and acknowledged (ACK Range); see Section 19.3.1.

ECN Counts: The three ECN Counts; see Section 19.3.2.

19.3.1. ACK Ranges

Each ACK Range consists of alternating Gap and ACK Range Length values in descending packet number order. ACK Ranges can be repeated. The number of Gap and ACK Range Length values is determined by the ACK Range Count field; one of each value is present for each value in the ACK Range Count field.

ACK Ranges are structured as shown in Figure 26.

```
ACK Range {  
    Gap (i),  
    ACK Range Length (i),  
}
```

Figure 26: ACK Ranges

The fields that form each ACK Range are:

Gap: A variable-length integer indicating the number of contiguous unacknowledged packets preceding the packet number one lower than the smallest in the preceding ACK Range.

ACK Range Length: A variable-length integer indicating the number of

contiguous acknowledged packets preceding the largest packet number, as determined by the preceding Gap.

Gap and ACK Range Length values use a relative integer encoding for efficiency. Though each encoded value is positive, the values are subtracted, so that each ACK Range describes progressively lower-numbered packets.

Each ACK Range acknowledges a contiguous range of packets by indicating the number of acknowledged packets that precede the largest packet number in that range. A value of zero indicates that only the largest packet number is acknowledged. Larger ACK Range values indicate a larger range, with corresponding lower values for the smallest packet number in the range. Thus, given a largest packet number for the range, the smallest value is determined by the formula:

$$\text{smallest} = \text{largest} - \text{ack_range}$$

An ACK Range acknowledges all packets between the smallest packet number and the largest, inclusive.

The largest value for an ACK Range is determined by cumulatively subtracting the size of all preceding ACK Range Lengths and Gaps.

Each Gap indicates a range of packets that are not being acknowledged. The number of packets in the gap is one higher than the encoded value of the Gap field.

The value of the Gap field establishes the largest packet number value for the subsequent ACK Range using the following formula:

$$\text{largest} = \text{previous_smallest} - \text{gap} - 2$$

If any computed packet number is negative, an endpoint MUST generate a connection error of type `FRAME_ENCODING_ERROR`.

19.3.2. ECN Counts

The ACK frame uses the least significant bit of the type value (that is, type `0x03`) to indicate ECN feedback and report receipt of QUIC packets with associated ECN codepoints of `ECT(0)`, `ECT(1)`, or `CE` in the packet's IP header. ECN Counts are only present when the ACK frame type is `0x03`.

When present, there are 3 ECN counts, as shown in Figure 27.

```
ECN Counts {  
    ECT0 Count (i),  
    ECT1 Count (i),  
    ECN-CE Count (i),  
}
```

Figure 27: ECN Count Format

The three ECN Counts are:

ECT0 Count: A variable-length integer representing the total number of packets received with the ECT(0) codepoint in the packet number space of the ACK frame.

ECT1 Count: A variable-length integer representing the total number of packets received with the ECT(1) codepoint in the packet number space of the ACK frame.

CE Count: A variable-length integer representing the total number of packets received with the CE codepoint in the packet number space of the ACK frame.

ECN counts are maintained separately for each packet number space.

19.4. RESET_STREAM Frames

An endpoint uses a RESET_STREAM frame (type=0x04) to abruptly terminate the sending part of a stream.

After sending a RESET_STREAM, an endpoint ceases transmission and retransmission of STREAM frames on the identified stream. A receiver of RESET_STREAM can discard any data that it already received on that stream.

An endpoint that receives a RESET_STREAM frame for a send-only stream MUST terminate the connection with error STREAM_STATE_ERROR.

RESET_STREAM frames are formatted as shown in Figure 28.

```
RESET_STREAM Frame {  
    Type (i) = 0x04,  
    Stream ID (i),  
    Application Protocol Error Code (i),  
    Final Size (i),  
}
```

Figure 28: RESET_STREAM Frame Format

RESET_STREAM frames contain the following fields:

Stream ID: A variable-length integer encoding of the Stream ID of the stream being terminated.

Application Protocol Error Code: A variable-length integer containing the application protocol error code (see Section 20.2) that indicates why the stream is being closed.

Final Size: A variable-length integer indicating the final size of the stream by the RESET_STREAM sender, in unit of bytes; see Section 4.5.

19.5. STOP_SENDING Frames

An endpoint uses a STOP_SENDING frame (type=0x05) to communicate that incoming data is being discarded on receipt at application request. STOP_SENDING requests that a peer cease transmission on a stream.

A STOP_SENDING frame can be sent for streams in the Recv or Size Known states; see Section 3.1. Receiving a STOP_SENDING frame for a locally-initiated stream that has not yet been created MUST be treated as a connection error of type STREAM_STATE_ERROR. An endpoint that receives a STOP_SENDING frame for a receive-only stream MUST terminate the connection with error STREAM_STATE_ERROR.

STOP_SENDING frames are formatted as shown in Figure 29.

```
STOP_SENDING Frame {  
  Type (i) = 0x05,  
  Stream ID (i),  
  Application Protocol Error Code (i),  
}
```

Figure 29: STOP_SENDING Frame Format

STOP_SENDING frames contain the following fields:

Stream ID: A variable-length integer carrying the Stream ID of the stream being ignored.

Application Protocol Error Code: A variable-length integer containing the application-specified reason the sender is ignoring the stream; see Section 20.2.

19.6. CRYPTO Frames

A CRYPTO frame (type=0x06) is used to transmit cryptographic handshake messages. It can be sent in all packet types except 0-RTT. The CRYPTO frame offers the cryptographic protocol an in-order stream of bytes. CRYPTO frames are functionally identical to STREAM frames, except that they do not bear a stream identifier; they are not flow controlled; and they do not carry markers for optional offset, optional length, and the end of the stream.

CRYPTO frames are formatted as shown in Figure 30.

```
CRYPTO Frame {  
    Type (i) = 0x06,  
    Offset (i),  
    Length (i),  
    Crypto Data (...),  
}
```

Figure 30: CRYPTO Frame Format

CRYPTO frames contain the following fields:

Offset: A variable-length integer specifying the byte offset in the stream for the data in this CRYPTO frame.

Length: A variable-length integer specifying the length of the Crypto Data field in this CRYPTO frame.

Crypto Data: The cryptographic message data.

There is a separate flow of cryptographic handshake data in each encryption level, each of which starts at an offset of 0. This implies that each encryption level is treated as a separate CRYPTO stream of data.

The largest offset delivered on a stream - the sum of the offset and data length - cannot exceed $2^{62}-1$. Receipt of a frame that exceeds this limit MUST be treated as a connection error of type `FRAME_ENCODING_ERROR` or `CRYPTO_BUFFER_EXCEEDED`.

Unlike STREAM frames, which include a Stream ID indicating to which stream the data belongs, the CRYPTO frame carries data for a single stream per encryption level. The stream does not have an explicit end, so CRYPTO frames do not have a FIN bit.

19.7. NEW_TOKEN Frames

A server sends a NEW_TOKEN frame (type=0x07) to provide the client with a token to send in the header of an Initial packet for a future connection.

NEW_TOKEN frames are formatted as shown in Figure 31.

```
NEW_TOKEN Frame {  
    Type (i) = 0x07,  
    Token Length (i),  
    Token (..),  
}
```

Figure 31: NEW_TOKEN Frame Format

NEW_TOKEN frames contain the following fields:

Token Length: A variable-length integer specifying the length of the token in bytes.

Token: An opaque blob that the client can use with a future Initial packet. The token **MUST NOT** be empty. A client **MUST** treat receipt of a NEW_TOKEN frame with an empty Token field as a connection error of type FRAME_ENCODING_ERROR.

A client might receive multiple NEW_TOKEN frames that contain the same token value if packets containing the frame are incorrectly determined to be lost. Clients are responsible for discarding duplicate values, which might be used to link connection attempts; see Section 8.1.3.

Clients **MUST NOT** send NEW_TOKEN frames. A server **MUST** treat receipt of a NEW_TOKEN frame as a connection error of type PROTOCOL_VIOLATION.

19.8. STREAM Frames

STREAM frames implicitly create a stream and carry stream data. The STREAM frame Type field takes the form 0b000001XXX (or the set of values from 0x08 to 0x0f). The three low-order bits of the frame type determine the fields that are present in the frame:

- * The OFF bit (0x04) in the frame type is set to indicate that there is an Offset field present. When set to 1, the Offset field is present. When set to 0, the Offset field is absent and the Stream Data starts at an offset of 0 (that is, the frame contains the first bytes of the stream, or the end of a stream that includes no data).
- * The LEN bit (0x02) in the frame type is set to indicate that there is a Length field present. If this bit is set to 0, the Length field is absent and the Stream Data field extends to the end of the packet. If this bit is set to 1, the Length field is present.
- * The FIN bit (0x01) indicates that the frame marks the end of the stream. The final size of the stream is the sum of the offset and the length of this frame.

An endpoint MUST terminate the connection with error `STREAM_STATE_ERROR` if it receives a `STREAM` frame for a locally-initiated stream that has not yet been created, or for a send-only stream.

`STREAM` frames are formatted as shown in Figure 32.

```
STREAM Frame {  
    Type (i) = 0x08..0x0f,  
    Stream ID (i),  
    [Offset (i)],  
    [Length (i)],  
    Stream Data (..),  
}
```

Figure 32: `STREAM` Frame Format

`STREAM` frames contain the following fields:

Stream ID: A variable-length integer indicating the stream ID of the stream; see Section 2.1.

Offset: A variable-length integer specifying the byte offset in the stream for the data in this `STREAM` frame. This field is present when the OFF bit is set to 1. When the Offset field is absent, the offset is 0.

Length: A variable-length integer specifying the length of the Stream Data field in this `STREAM` frame. This field is present when the LEN bit is set to 1. When the LEN bit is set to 0, the Stream Data field consumes all the remaining bytes in the packet.

Stream Data: The bytes from the designated stream to be delivered.

When a Stream Data field has a length of 0, the offset in the STREAM frame is the offset of the next byte that would be sent.

The first byte in the stream has an offset of 0. The largest offset delivered on a stream - the sum of the offset and data length - cannot exceed $2^{62}-1$, as it is not possible to provide flow control credit for that data. Receipt of a frame that exceeds this limit MUST be treated as a connection error of type FRAME_ENCODING_ERROR or FLOW_CONTROL_ERROR.

19.9. MAX_DATA Frames

A MAX_DATA frame (type=0x10) is used in flow control to inform the peer of the maximum amount of data that can be sent on the connection as a whole.

MAX_DATA frames are formatted as shown in Figure 33.

```
MAX_DATA Frame {  
    Type (i) = 0x10,  
    Maximum Data (i),  
}
```

Figure 33: MAX_DATA Frame Format

MAX_DATA frames contain the following field:

Maximum Data: A variable-length integer indicating the maximum amount of data that can be sent on the entire connection, in units of bytes.

All data sent in STREAM frames counts toward this limit. The sum of the final sizes on all streams - including streams in terminal states - MUST NOT exceed the value advertised by a receiver. An endpoint MUST terminate a connection with a FLOW_CONTROL_ERROR error if it receives more data than the maximum data value that it has sent. This includes violations of remembered limits in Early Data; see Section 7.4.1.

19.10. MAX_STREAM_DATA Frames

A MAX_STREAM_DATA frame (type=0x11) is used in flow control to inform a peer of the maximum amount of data that can be sent on a stream.

A `MAX_STREAM_DATA` frame can be sent for streams in the Recv state; see Section 3.1. Receiving a `MAX_STREAM_DATA` frame for a locally-initiated stream that has not yet been created **MUST** be treated as a connection error of type `STREAM_STATE_ERROR`. An endpoint that receives a `MAX_STREAM_DATA` frame for a receive-only stream **MUST** terminate the connection with error `STREAM_STATE_ERROR`.

`MAX_STREAM_DATA` frames are formatted as shown in Figure 34.

```
MAX_STREAM_DATA Frame {  
    Type (i) = 0x11,  
    Stream ID (i),  
    Maximum Stream Data (i),  
}
```

Figure 34: `MAX_STREAM_DATA` Frame Format

`MAX_STREAM_DATA` frames contain the following fields:

Stream ID: The stream ID of the stream that is affected encoded as a variable-length integer.

Maximum Stream Data: A variable-length integer indicating the maximum amount of data that can be sent on the identified stream, in units of bytes.

When counting data toward this limit, an endpoint accounts for the largest received offset of data that is sent or received on the stream. Loss or reordering can mean that the largest received offset on a stream can be greater than the total size of data received on that stream. Receiving `STREAM` frames might not increase the largest received offset.

The data sent on a stream **MUST NOT** exceed the largest maximum stream data value advertised by the receiver. An endpoint **MUST** terminate a connection with a `FLOW_CONTROL_ERROR` error if it receives more data than the largest maximum stream data that it has sent for the affected stream. This includes violations of remembered limits in Early Data; see Section 7.4.1.

19.11. `MAX_STREAMS` Frames

A `MAX_STREAMS` frame (type=0x12 or 0x13) inform the peer of the cumulative number of streams of a given type it is permitted to open. A `MAX_STREAMS` frame with a type of 0x12 applies to bidirectional streams, and a `MAX_STREAMS` frame with a type of 0x13 applies to unidirectional streams.

MAX_STREAMS frames are formatted as shown in Figure 35;

```
MAX_STREAMS Frame {  
    Type (i) = 0x12..0x13,  
    Maximum Streams (i),  
}
```

Figure 35: MAX_STREAMS Frame Format

MAX_STREAMS frames contain the following field:

Maximum Streams: A count of the cumulative number of streams of the corresponding type that can be opened over the lifetime of the connection. This value cannot exceed 2^{60} , as it is not possible to encode stream IDs larger than $2^{62}-1$. Receipt of a frame that permits opening of a stream larger than this limit MUST be treated as a FRAME_ENCODING_ERROR.

Loss or reordering can cause a MAX_STREAMS frame to be received that state a lower stream limit than an endpoint has previously received. MAX_STREAMS frames that do not increase the stream limit MUST be ignored.

An endpoint MUST NOT open more streams than permitted by the current stream limit set by its peer. For instance, a server that receives a unidirectional stream limit of 3 is permitted to open stream 3, 7, and 11, but not stream 15. An endpoint MUST terminate a connection with a STREAM_LIMIT_ERROR error if a peer opens more streams than was permitted. This includes violations of remembered limits in Early Data; see Section 7.4.1.

Note that these frames (and the corresponding transport parameters) do not describe the number of streams that can be opened concurrently. The limit includes streams that have been closed as well as those that are open.

19.12. DATA_BLOCKED Frames

A sender SHOULD send a DATA_BLOCKED frame (type=0x14) when it wishes to send data, but is unable to do so due to connection-level flow control; see Section 4. DATA_BLOCKED frames can be used as input to tuning of flow control algorithms; see Section 4.2.

DATA_BLOCKED frames are formatted as shown in Figure 36.

```
DATA_BLOCKED Frame {  
  Type (i) = 0x14,  
  Maximum Data (i),  
}
```

Figure 36: DATA_BLOCKED Frame Format

DATA_BLOCKED frames contain the following field:

Maximum Data: A variable-length integer indicating the connection-level limit at which blocking occurred.

19.13. STREAM_DATA_BLOCKED Frames

A sender SHOULD send a STREAM_DATA_BLOCKED frame (type=0x15) when it wishes to send data, but is unable to do so due to stream-level flow control. This frame is analogous to DATA_BLOCKED (Section 19.12).

An endpoint that receives a STREAM_DATA_BLOCKED frame for a send-only stream MUST terminate the connection with error STREAM_STATE_ERROR.

STREAM_DATA_BLOCKED frames are formatted as shown in Figure 37.

```
STREAM_DATA_BLOCKED Frame {  
  Type (i) = 0x15,  
  Stream ID (i),  
  Maximum Stream Data (i),  
}
```

Figure 37: STREAM_DATA_BLOCKED Frame Format

STREAM_DATA_BLOCKED frames contain the following fields:

Stream ID: A variable-length integer indicating the stream that is blocked due to flow control.

Maximum Stream Data: A variable-length integer indicating the offset of the stream at which the blocking occurred.

19.14. STREAMS_BLOCKED Frames

A sender SHOULD send a STREAMS_BLOCKED frame (type=0x16 or 0x17) when it wishes to open a stream, but is unable to do so due to the maximum stream limit set by its peer; see Section 19.11. A STREAMS_BLOCKED frame of type 0x16 is used to indicate reaching the bidirectional stream limit, and a STREAMS_BLOCKED frame of type 0x17 is used to indicate reaching the unidirectional stream limit.

A STREAMS_BLOCKED frame does not open the stream, but informs the peer that a new stream was needed and the stream limit prevented the creation of the stream.

STREAMS_BLOCKED frames are formatted as shown in Figure 38.

```
STREAMS_BLOCKED Frame {  
    Type (i) = 0x16..0x17,  
    Maximum Streams (i),  
}
```

Figure 38: STREAMS_BLOCKED Frame Format

STREAMS_BLOCKED frames contain the following field:

Maximum Streams: A variable-length integer indicating the maximum number of streams allowed at the time the frame was sent. This value cannot exceed 2^{60} , as it is not possible to encode stream IDs larger than $2^{62}-1$. Receipt of a frame that encodes a larger stream ID MUST be treated as a STREAM_LIMIT_ERROR or a FRAME_ENCODING_ERROR.

19.15. NEW_CONNECTION_ID Frames

An endpoint sends a NEW_CONNECTION_ID frame (type=0x18) to provide its peer with alternative connection IDs that can be used to break linkability when migrating connections; see Section 9.5.

NEW_CONNECTION_ID frames are formatted as shown in Figure 39.

```
NEW_CONNECTION_ID Frame {  
    Type (i) = 0x18,  
    Sequence Number (i),  
    Retire Prior To (i),  
    Length (8),  
    Connection ID (8..160),  
    Stateless Reset Token (128),  
}
```

Figure 39: NEW_CONNECTION_ID Frame Format

NEW_CONNECTION_ID frames contain the following fields:

Sequence Number: The sequence number assigned to the connection ID by the sender, encoded as a variable-length integer; see Section 5.1.1.

Retire Prior To: A variable-length integer indicating which

connection IDs should be retired; see Section 5.1.2.

Length: An 8-bit unsigned integer containing the length of the connection ID. Values less than 1 and greater than 20 are invalid and MUST be treated as a connection error of type `FRAME_ENCODING_ERROR`.

Connection ID: A connection ID of the specified length.

Stateless Reset Token: A 128-bit value that will be used for a stateless reset when the associated connection ID is used; see Section 10.3.

An endpoint MUST NOT send this frame if it currently requires that its peer send packets with a zero-length Destination Connection ID. Changing the length of a connection ID to or from zero-length makes it difficult to identify when the value of the connection ID changed. An endpoint that is sending packets with a zero-length Destination Connection ID MUST treat receipt of a `NEW_CONNECTION_ID` frame as a connection error of type `PROTOCOL_VIOLATION`.

Transmission errors, timeouts and retransmissions might cause the same `NEW_CONNECTION_ID` frame to be received multiple times. Receipt of the same frame multiple times MUST NOT be treated as a connection error. A receiver can use the sequence number supplied in the `NEW_CONNECTION_ID` frame to handle receiving the same `NEW_CONNECTION_ID` frame multiple times.

If an endpoint receives a `NEW_CONNECTION_ID` frame that repeats a previously issued connection ID with a different Stateless Reset Token or a different sequence number, or if a sequence number is used for different connection IDs, the endpoint MAY treat that receipt as a connection error of type `PROTOCOL_VIOLATION`.

The `Retire Prior To` field applies to connection IDs established during connection setup and the `preferred_address` transport parameter; see Section 5.1.2. The `Retire Prior To` field MUST be less than or equal to the `Sequence Number` field. Receiving a value greater than the `Sequence Number` MUST be treated as a connection error of type `FRAME_ENCODING_ERROR`.

Once a sender indicates a `Retire Prior To` value, smaller values sent in subsequent `NEW_CONNECTION_ID` frames have no effect. A receiver MUST ignore any `Retire Prior To` fields that do not increase the largest received `Retire Prior To` value.

An endpoint that receives a `NEW_CONNECTION_ID` frame with a sequence number smaller than the `Retire Prior To` field of a previously received `NEW_CONNECTION_ID` frame MUST send a corresponding `RETIRE_CONNECTION_ID` frame that retires the newly received connection ID, unless it has already done so for that sequence number.

19.16. `RETIRE_CONNECTION_ID` Frames

An endpoint sends a `RETIRE_CONNECTION_ID` frame (type=0x19) to indicate that it will no longer use a connection ID that was issued by its peer. This includes the connection ID provided during the handshake. Sending a `RETIRE_CONNECTION_ID` frame also serves as a request to the peer to send additional connection IDs for future use; see Section 5.1. New connection IDs can be delivered to a peer using the `NEW_CONNECTION_ID` frame (Section 19.15).

Retiring a connection ID invalidates the stateless reset token associated with that connection ID.

`RETIRE_CONNECTION_ID` frames are formatted as shown in Figure 40.

```
RETIRE_CONNECTION_ID Frame {  
    Type (i) = 0x19,  
    Sequence Number (i),  
}
```

Figure 40: `RETIRE_CONNECTION_ID` Frame Format

`RETIRE_CONNECTION_ID` frames contain the following field:

Sequence Number: The sequence number of the connection ID being retired; see Section 5.1.2.

Receipt of a `RETIRE_CONNECTION_ID` frame containing a sequence number greater than any previously sent to the peer MUST be treated as a connection error of type `PROTOCOL_VIOLATION`.

The sequence number specified in a `RETIRE_CONNECTION_ID` frame MUST NOT refer to the Destination Connection ID field of the packet in which the frame is contained. The peer MAY treat this as a connection error of type `PROTOCOL_VIOLATION`.

An endpoint cannot send this frame if it was provided with a zero-length connection ID by its peer. An endpoint that provides a zero-length connection ID MUST treat receipt of a `RETIRE_CONNECTION_ID` frame as a connection error of type `PROTOCOL_VIOLATION`.

19.17. PATH_CHALLENGE Frames

Endpoints can use PATH_CHALLENGE frames (type=0x1a) to check reachability to the peer and for path validation during connection migration.

PATH_CHALLENGE frames are formatted as shown in Figure 41.

```
PATH_CHALLENGE Frame {  
    Type (i) = 0x1a,  
    Data (64),  
}
```

Figure 41: PATH_CHALLENGE Frame Format

PATH_CHALLENGE frames contain the following field:

Data: This 8-byte field contains arbitrary data.

Including 64 bits of entropy in a PATH_CHALLENGE frame ensures that it is easier to receive the packet than it is to guess the value correctly.

The recipient of this frame MUST generate a PATH_RESPONSE frame (Section 19.18) containing the same Data.

19.18. PATH_RESPONSE Frames

A PATH_RESPONSE frame (type=0x1b) is sent in response to a PATH_CHALLENGE frame.

PATH_RESPONSE frames are formatted as shown in Figure 42, which is identical to the PATH_CHALLENGE frame (Section 19.17).

```
PATH_RESPONSE Frame {  
    Type (i) = 0x1b,  
    Data (64),  
}
```

Figure 42: PATH_RESPONSE Frame Format

If the content of a PATH_RESPONSE frame does not match the content of a PATH_CHALLENGE frame previously sent by the endpoint, the endpoint MAY generate a connection error of type PROTOCOL_VIOLATION.

19.19. CONNECTION_CLOSE Frames

An endpoint sends a CONNECTION_CLOSE frame (type=0x1c or 0x1d) to notify its peer that the connection is being closed. The CONNECTION_CLOSE with a frame type of 0x1c is used to signal errors at only the QUIC layer, or the absence of errors (with the NO_ERROR code). The CONNECTION_CLOSE frame with a type of 0x1d is used to signal an error with the application that uses QUIC.

If there are open streams that have not been explicitly closed, they are implicitly closed when the connection is closed.

CONNECTION_CLOSE frames are formatted as shown in Figure 43.

```
CONNECTION_CLOSE Frame {  
    Type (i) = 0x1c..0x1d,  
    Error Code (i),  
    [Frame Type (i)],  
    Reason Phrase Length (i),  
    Reason Phrase (...),  
}
```

Figure 43: CONNECTION_CLOSE Frame Format

CONNECTION_CLOSE frames contain the following fields:

Error Code: A variable-length integer error code that indicates the reason for closing this connection. A CONNECTION_CLOSE frame of type 0x1c uses codes from the space defined in Section 20.1. A CONNECTION_CLOSE frame of type 0x1d uses codes from the application protocol error code space; see Section 20.2.

Frame Type: A variable-length integer encoding the type of frame that triggered the error. A value of 0 (equivalent to the mention of the PADDING frame) is used when the frame type is unknown. The application-specific variant of CONNECTION_CLOSE (type 0x1d) does not include this field.

Reason Phrase Length: A variable-length integer specifying the length of the reason phrase in bytes. Because a CONNECTION_CLOSE frame cannot be split between packets, any limits on packet size will also limit the space available for a reason phrase.

Reason Phrase: Additional diagnostic information for the closure.

This can be zero length if the sender chooses not to give details beyond the Error Code. This SHOULD be a UTF-8 encoded string [RFC3629], though the frame does not carry information, such as language tags, that would aid comprehension by any entity other than the one that created the text.

The application-specific variant of CONNECTION_CLOSE (type 0x1d) can only be sent using 0-RTT or 1-RTT packets; see Section 12.5. When an application wishes to abandon a connection during the handshake, an endpoint can send a CONNECTION_CLOSE frame (type 0x1c) with an error code of APPLICATION_ERROR in an Initial or a Handshake packet.

19.20. HANDSHAKE_DONE Frames

The server uses a HANDSHAKE_DONE frame (type=0x1e) to signal confirmation of the handshake to the client.

HANDSHAKE_DONE frames are formatted as shown in Figure 44, which shows that HANDSHAKE_DONE frames have no content.

```
HANDSHAKE_DONE Frame {  
    Type (i) = 0x1e,  
}
```

Figure 44: HANDSHAKE_DONE Frame Format

A HANDSHAKE_DONE frame can only be sent by the server. Servers MUST NOT send a HANDSHAKE_DONE frame before completing the handshake. A server MUST treat receipt of a HANDSHAKE_DONE frame as a connection error of type PROTOCOL_VIOLATION.

19.21. Extension Frames

QUIC frames do not use a self-describing encoding. An endpoint therefore needs to understand the syntax of all frames before it can successfully process a packet. This allows for efficient encoding of frames, but it means that an endpoint cannot send a frame of a type that is unknown to its peer.

An extension to QUIC that wishes to use a new type of frame MUST first ensure that a peer is able to understand the frame. An endpoint can use a transport parameter to signal its willingness to receive extension frame types. One transport parameter can indicate support for one or more extension frame types.

Extensions that modify or replace core protocol functionality (including frame types) will be difficult to combine with other extensions that modify or replace the same functionality unless the

behavior of the combination is explicitly defined. Such extensions SHOULD define their interaction with previously-defined extensions modifying the same protocol components.

Extension frames MUST be congestion controlled and MUST cause an ACK frame to be sent. The exception is extension frames that replace or supplement the ACK frame. Extension frames are not included in flow control unless specified in the extension.

An IANA registry is used to manage the assignment of frame types; see Section 22.4.

20. Error Codes

QUIC transport error codes and application error codes are 62-bit unsigned integers.

20.1. Transport Error Codes

This section lists the defined QUIC transport error codes that can be used in a CONNECTION_CLOSE frame with a type of 0x1c. These errors apply to the entire connection.

NO_ERROR (0x0): An endpoint uses this with CONNECTION_CLOSE to signal that the connection is being closed abruptly in the absence of any error.

INTERNAL_ERROR (0x1): The endpoint encountered an internal error and cannot continue with the connection.

CONNECTION_REFUSED (0x2): The server refused to accept a new connection.

FLOW_CONTROL_ERROR (0x3): An endpoint received more data than it permitted in its advertised data limits; see Section 4.

STREAM_LIMIT_ERROR (0x4): An endpoint received a frame for a stream identifier that exceeded its advertised stream limit for the corresponding stream type.

STREAM_STATE_ERROR (0x5): An endpoint received a frame for a stream that was not in a state that permitted that frame; see Section 3.

FINAL_SIZE_ERROR (0x6): An endpoint received a STREAM frame

containing data that exceeded the previously established final size. Or an endpoint received a STREAM frame or a RESET_STREAM frame containing a final size that was lower than the size of stream data that was already received. Or an endpoint received a STREAM frame or a RESET_STREAM frame containing a different final size to the one already established.

FRAME_ENCODING_ERROR (0x7): An endpoint received a frame that was badly formatted. For instance, a frame of an unknown type, or an ACK frame that has more acknowledgment ranges than the remainder of the packet could carry.

TRANSPORT_PARAMETER_ERROR (0x8): An endpoint received transport parameters that were badly formatted, included an invalid value, omitted a mandatory transport parameter, included a forbidden transport parameter, or were otherwise in error.

CONNECTION_ID_LIMIT_ERROR (0x9): The number of connection IDs provided by the peer exceeds the advertised `active_connection_id_limit`.

PROTOCOL_VIOLATION (0xa): An endpoint detected an error with protocol compliance that was not covered by more specific error codes.

INVALID_TOKEN (0xb): A server received a client Initial that contained an invalid Token field.

APPLICATION_ERROR (0xc): The application or application protocol caused the connection to be closed.

CRYPTO_BUFFER_EXCEEDED (0xd): An endpoint has received more data in CRYPTO frames than it can buffer.

KEY_UPDATE_ERROR (0xe): An endpoint detected errors in performing key updates; see Section 6 of [QUIC-TLS].

AEAD_LIMIT_REACHED (0xf): An endpoint has reached the confidentiality or integrity limit for the AEAD algorithm used by the given connection.

NO_VIABLE_PATH (0x10): An endpoint has determined that the network path is incapable of supporting QUIC. An endpoint is unlikely to receive CONNECTION_CLOSE carrying this code except when the path does not support a large enough MTU.

CRYPTO_ERROR (0x1XX): The cryptographic handshake failed. A range

of 256 values is reserved for carrying error codes specific to the cryptographic handshake that is used. Codes for errors occurring when TLS is used for the crypto handshake are described in Section 4.8 of [QUIC-TLS].

See Section 22.5 for details of registering new error codes.

In defining these error codes, several principles are applied. Error conditions that might require specific action on the part of a recipient are given unique codes. Errors that represent common conditions are given specific codes. Absent either of these conditions, error codes are used to identify a general function of the stack, like flow control or transport parameter handling. Finally, generic errors are provided for conditions where implementations are unable or unwilling to use more specific codes.

20.2. Application Protocol Error Codes

The management of application error codes is left to application protocols. Application protocol error codes are used for the RESET_STREAM frame (Section 19.4), the STOP_SENDING frame (Section 19.5), and the CONNECTION_CLOSE frame with a type of 0x1d (Section 19.19).

21. Security Considerations

The goal of QUIC is to provide a secure transport connection. Section 21.1 provides an overview of those properties; subsequent sections discuss constraints and caveats regarding these properties, including descriptions of known attacks and countermeasures.

21.1. Overview of Security Properties

A complete security analysis of QUIC is outside the scope of this document. This section provides an informal description of the desired security properties as an aid to implementors and to help guide protocol analysis.

QUIC assumes the threat model described in [SEC-CONS] and provides protections against many of the attacks that arise from that model.

For this purpose, attacks are divided into passive and active attacks. Passive attackers have the capability to read packets from the network, while active attackers also have the capability to write packets into the network. However, a passive attack could involve an attacker with the ability to cause a routing change or other modification in the path taken by packets that comprise a connection.

Attackers are additionally categorized as either on-path attackers or off-path attackers. An on-path attacker can read, modify, or remove any packet it observes such that it no longer reaches its destination, while an off-path attacker observes the packets, but cannot prevent the original packet from reaching its intended destination. Both types of attackers can also transmit arbitrary packets. This definition differs from that of Section 3.5 of [SEC-CONS] in that an off-path attacker is able to observe packets.

Properties of the handshake, protected packets, and connection migration are considered separately.

21.1.1.1. Handshake

The QUIC handshake incorporates the TLS 1.3 handshake and inherits the cryptographic properties described in Appendix E.1 of [TLS13]. Many of the security properties of QUIC depend on the TLS handshake providing these properties. Any attack on the TLS handshake could affect QUIC.

Any attack on the TLS handshake that compromises the secrecy or uniqueness of session keys, or the authentication of the participating peers, affects other security guarantees provided by QUIC that depend on those keys. For instance, migration (Section 9) depends on the efficacy of confidentiality protections, both for the negotiation of keys using the TLS handshake and for QUIC packet protection, to avoid linkability across network paths.

An attack on the integrity of the TLS handshake might allow an attacker to affect the selection of application protocol or QUIC version.

In addition to the properties provided by TLS, the QUIC handshake provides some defense against DoS attacks on the handshake.

21.1.1.1.1. Anti-Amplification

Address validation (Section 8) is used to verify that an entity that claims a given address is able to receive packets at that address. Address validation limits amplification attack targets to addresses for which an attacker can observe packets.

Prior to address validation, endpoints are limited in what they are able to send. Endpoints cannot send data toward an unvalidated address in excess of three times the data received from that address.

Note: The anti-amplification limit only applies when an endpoint

responds to packets received from an unvalidated address. The anti-amplification limit does not apply to clients when establishing a new connection or when initiating connection migration.

21.1.1.2. Server-Side DoS

Computing the server's first flight for a full handshake is potentially expensive, requiring both a signature and a key exchange computation. In order to prevent computational DoS attacks, the Retry packet provides a cheap token exchange mechanism that allows servers to validate a client's IP address prior to doing any expensive computations at the cost of a single round trip. After a successful handshake, servers can issue new tokens to a client, which will allow new connection establishment without incurring this cost.

21.1.1.3. On-Path Handshake Termination

An on-path or off-path attacker can force a handshake to fail by replacing or racing Initial packets. Once valid Initial packets have been exchanged, subsequent Handshake packets are protected with the handshake keys and an on-path attacker cannot force handshake failure other than by dropping packets to cause endpoints to abandon the attempt.

An on-path attacker can also replace the addresses of packets on either side and therefore cause the client or server to have an incorrect view of the remote addresses. Such an attack is indistinguishable from the functions performed by a NAT.

21.1.1.4. Parameter Negotiation

The entire handshake is cryptographically protected, with the Initial packets being encrypted with per-version keys and the Handshake and later packets being encrypted with keys derived from the TLS key exchange. Further, parameter negotiation is folded into the TLS transcript and thus provides the same integrity guarantees as ordinary TLS negotiation. An attacker can observe the client's transport parameters (as long as it knows the version-specific salt) but cannot observe the server's transport parameters and cannot influence parameter negotiation.

Connection IDs are unencrypted but integrity protected in all packets.

This version of QUIC does not incorporate a version negotiation mechanism; implementations of incompatible versions will simply fail to establish a connection.

21.1.2. Protected Packets

Packet protection (Section 12.1) applies authenticated encryption to all packets except Version Negotiation packets, though Initial and Retry packets have limited protection due to the use of version-specific keying material; see [QUIC-TLS] for more details. This section considers passive and active attacks against protected packets.

Both on-path and off-path attackers can mount a passive attack in which they save observed packets for an offline attack against packet protection at a future time; this is true for any observer of any packet on any network.

A blind attacker, one who injects packets without being able to observe valid packets for a connection, is unlikely to be successful, since packet protection ensures that valid packets are only generated by endpoints that possess the key material established during the handshake; see Section 7 and Section 21.1.1. Similarly, any active attacker that observes packets and attempts to insert new data or modify existing data in those packets should not be able to generate packets deemed valid by the receiving endpoint, other than Initial packets.

A spoofing attack, in which an active attacker rewrites unprotected parts of a packet that it forwards or injects, such as the source or destination address, is only effective if the attacker can forward packets to the original endpoint. Packet protection ensures that the packet payloads can only be processed by the endpoints that completed the handshake, and invalid packets are ignored by those endpoints.

An attacker can also modify the boundaries between packets and UDP datagrams, causing multiple packets to be coalesced into a single datagram, or splitting coalesced packets into multiple datagrams. Aside from datagrams containing Initial packets, which require padding, modification of how packets are arranged in datagrams has no functional effect on a connection, although it might change some performance characteristics.

21.1.3. Connection Migration

Connection Migration (Section 9) provides endpoints with the ability to transition between IP addresses and ports on multiple paths, using one path at a time for transmission and receipt of non-probing frames. Path validation (Section 8.2) establishes that a peer is both willing and able to receive packets sent on a particular path. This helps reduce the effects of address spoofing by limiting the number of packets sent to a spoofed address.

This section describes the intended security properties of connection migration under various types of DoS attacks.

21.1.3.1. On-Path Active Attacks

An attacker that can cause a packet it observes to no longer reach its intended destination is considered an on-path attacker. When an attacker is present between a client and server, endpoints are required to send packets through the attacker to establish connectivity on a given path.

An on-path attacker can:

- * Inspect packets
- * Modify IP and UDP packet headers
- * Inject new packets
- * Delay packets
- * Reorder packets
- * Drop packets
- * Split and merge datagrams along packet boundaries

An on-path attacker cannot:

- * Modify an authenticated portion of a packet and cause the recipient to accept that packet

An on-path attacker has the opportunity to modify the packets that it observes, however any modifications to an authenticated portion of a packet will cause it to be dropped by the receiving endpoint as invalid, as packet payloads are both authenticated and encrypted.

In the presence of an on-path attacker, QUIC aims to provide the following properties:

1. An on-path attacker can prevent use of a path for a connection, causing the connection to fail if it cannot use a different path that does not contain the attacker. This can be achieved by dropping all packets, modifying them so that they fail to decrypt, or other methods.

2. An on-path attacker can prevent migration to a new path for which the attacker is also on-path by causing path validation to fail on the new path.
3. An on-path attacker cannot prevent a client from migrating to a path for which the attacker is not on-path.
4. An on-path attacker can reduce the throughput of a connection by delaying packets or dropping them.
5. An on-path attacker cannot cause an endpoint to accept a packet for which it has modified an authenticated portion of that packet.

21.1.3.2. Off-Path Active Attacks

An off-path attacker is not directly on the path between a client and server, but could be able to obtain copies of some or all packets sent between the client and the server. It is also able to send copies of those packets to either endpoint.

An off-path attacker can:

- * Inspect packets
- * Inject new packets
- * Reorder injected packets

An off-path attacker cannot:

- * Modify packets sent by endpoints
- * Delay packets
- * Drop packets
- * Reorder original packets

An off-path attacker can create modified copies of packets that it has observed and inject those copies into the network, potentially with spoofed source and destination addresses.

For the purposes of this discussion, it is assumed that an off-path attacker has the ability to inject a modified copy of a packet into the network that will reach the destination endpoint prior to the arrival of the original packet observed by the attacker. In other words, an attacker has the ability to consistently "win" a race with the legitimate packets between the endpoints, potentially causing the original packet to be ignored by the recipient.

It is also assumed that an attacker has the resources necessary to affect NAT state, potentially both causing an endpoint to lose its NAT binding, and an attacker to obtain the same port for use with its traffic.

In the presence of an off-path attacker, QUIC aims to provide the following properties:

1. An off-path attacker can race packets and attempt to become a "limited" on-path attacker.
2. An off-path attacker can cause path validation to succeed for forwarded packets with the source address listed as the off-path attacker as long as it can provide improved connectivity between the client and the server.
3. An off-path attacker cannot cause a connection to close once the handshake has completed.
4. An off-path attacker cannot cause migration to a new path to fail if it cannot observe the new path.
5. An off-path attacker can become a limited on-path attacker during migration to a new path for which it is also an off-path attacker.
6. An off-path attacker can become a limited on-path attacker by affecting shared NAT state such that it sends packets to the server from the same IP address and port that the client originally used.

21.1.3.3. Limited On-Path Active Attacks

A limited on-path attacker is an off-path attacker that has offered improved routing of packets by duplicating and forwarding original packets between the server and the client, causing those packets to arrive before the original copies such that the original packets are dropped by the destination endpoint.

A limited on-path attacker differs from an on-path attacker in that it is not on the original path between endpoints, and therefore the original packets sent by an endpoint are still reaching their destination. This means that a future failure to route copied packets to the destination faster than their original path will not prevent the original packets from reaching the destination.

A limited on-path attacker can:

- * Inspect packets
- * Inject new packets
- * Modify unencrypted packet headers
- * Reorder packets

A limited on-path attacker cannot:

- * Delay packets so that they arrive later than packets sent on the original path
- * Drop packets
- * Modify the authenticated and encrypted portion of a packet and cause the recipient to accept that packet

A limited on-path attacker can only delay packets up to the point that the original packets arrive before the duplicate packets, meaning that it cannot offer routing with worse latency than the original path. If a limited on-path attacker drops packets, the original copy will still arrive at the destination endpoint.

In the presence of a limited on-path attacker, QUIC aims to provide the following properties:

1. A limited on-path attacker cannot cause a connection to close once the handshake has completed.
2. A limited on-path attacker cannot cause an idle connection to close if the client is first to resume activity.
3. A limited on-path attacker can cause an idle connection to be deemed lost if the server is the first to resume activity.

Note that these guarantees are the same guarantees provided for any NAT, for the same reasons.

21.2. Handshake Denial of Service

As an encrypted and authenticated transport QUIC provides a range of protections against denial of service. Once the cryptographic handshake is complete, QUIC endpoints discard most packets that are not authenticated, greatly limiting the ability of an attacker to interfere with existing connections.

Once a connection is established QUIC endpoints might accept some unauthenticated ICMP packets (see Section 14.2.1), but the use of these packets is extremely limited. The only other type of packet that an endpoint might accept is a stateless reset (Section 10.3), which relies on the token being kept secret until it is used.

During the creation of a connection, QUIC only provides protection against attack from off the network path. All QUIC packets contain proof that the recipient saw a preceding packet from its peer.

Addresses cannot change during the handshake, so endpoints can discard packets that are received on a different network path.

The Source and Destination Connection ID fields are the primary means of protection against off-path attack during the handshake; see Section 8.1. These are required to match those set by a peer. Except for an Initial and stateless reset packets, an endpoint only accepts packets that include a Destination Connection ID field that matches a value the endpoint previously chose. This is the only protection offered for Version Negotiation packets.

The Destination Connection ID field in an Initial packet is selected by a client to be unpredictable, which serves an additional purpose. The packets that carry the cryptographic handshake are protected with a key that is derived from this connection ID and a salt specific to the QUIC version. This allows endpoints to use the same process for authenticating packets that they receive as they use after the cryptographic handshake completes. Packets that cannot be authenticated are discarded. Protecting packets in this fashion provides a strong assurance that the sender of the packet saw the Initial packet and understood it.

These protections are not intended to be effective against an attacker that is able to receive QUIC packets prior to the connection being established. Such an attacker can potentially send packets that will be accepted by QUIC endpoints. This version of QUIC attempts to detect this sort of attack, but it expects that endpoints will fail to establish a connection rather than recovering. For the most part, the cryptographic handshake protocol [QUIC-TLS] is responsible for detecting tampering during the handshake.

Endpoints are permitted to use other methods to detect and attempt to recover from interference with the handshake. Invalid packets can be identified and discarded using other methods, but no specific method is mandated in this document.

21.3. Amplification Attack

An attacker might be able to receive an address validation token (Section 8) from a server and then release the IP address it used to acquire that token. At a later time, the attacker can initiate a 0-RTT connection with a server by spoofing this same address, which might now address a different (victim) endpoint. The attacker can thus potentially cause the server to send an initial congestion window's worth of data towards the victim.

Servers SHOULD provide mitigations for this attack by limiting the usage and lifetime of address validation tokens; see Section 8.1.3.

21.4. Optimistic ACK Attack

An endpoint that acknowledges packets it has not received might cause a congestion controller to permit sending at rates beyond what the network supports. An endpoint MAY skip packet numbers when sending packets to detect this behavior. An endpoint can then immediately close the connection with a connection error of type `PROTOCOL_VIOLATION`; see Section 10.2.

21.5. Request Forgery Attacks

A request forgery attack occurs where an endpoint causes its peer to issue a request towards a victim, with the request controlled by the endpoint. Request forgery attacks aim to provide an attacker with access to capabilities of its peer that might otherwise be unavailable to the attacker. For a networking protocol, a request forgery attack is often used to exploit any implicit authorization conferred on the peer by the victim due to the peer's location in the network.

For request forgery to be effective, an attacker needs to be able to influence what packets the peer sends and where these packets are sent. If an attacker can target a vulnerable service with a controlled payload, that service might perform actions that are attributed to the attacker's peer, but decided by the attacker.

For example, cross-site request forgery [CSRF] exploits on the Web cause a client to issue requests that include authorization cookies [COOKIE], allowing one site access to information and actions that are intended to be restricted to a different site.

As QUIC runs over UDP, the primary attack modality of concern is one where an attacker can select the address to which its peer sends UDP datagrams and can control some of the unprotected content of those packets. As much of the data sent by QUIC endpoints is protected, this includes control over ciphertext. An attack is successful if an attacker can cause a peer to send a UDP datagram to a host that will perform some action based on content in the datagram.

This section discusses ways in which QUIC might be used for request forgery attacks.

This section also describes limited countermeasures that can be implemented by QUIC endpoints. These mitigations can be employed unilaterally by a QUIC implementation or deployment, without potential targets for request forgery attacks taking action. However these countermeasures could be insufficient if UDP-based services do not properly authorize requests.

Because the migration attack described in Section 21.5.4 is quite powerful and does not have adequate countermeasures, QUIC server implementations should assume that attackers can cause them to generate arbitrary UDP payloads to arbitrary destinations. QUIC servers SHOULD NOT be deployed in networks that do not deploy ingress filtering [BCP38] and also have inadequately secured UDP endpoints.

Although it is not generally possible to ensure that clients are not co-located with vulnerable endpoints, this version of QUIC does not allow servers to migrate, thus preventing spoofed migration attacks on clients. Any future extension which allows server migration MUST also define countermeasures for forgery attacks.

21.5.1. Control Options for Endpoints

QUIC offers some opportunities for an attacker to influence or control where its peer sends UDP datagrams:

- * initial connection establishment (Section 7), where a server is able to choose where a client sends datagrams, for example by populating DNS records;
- * preferred addresses (Section 9.6), where a server is able to choose where a client sends datagrams;
- * spoofed connection migrations (Section 9.3.1), where a client is able to use source address spoofing to select where a server sends subsequent datagrams; and

- * spoofed packets that cause a server to send a Version Negotiation packet Section 21.5.5.

In all cases, the attacker can cause its peer to send datagrams to a victim that might not understand QUIC. That is, these packets are sent by the peer prior to address validation; see Section 8.

Outside of the encrypted portion of packets, QUIC offers an endpoint several options for controlling the content of UDP datagrams that its peer sends. The Destination Connection ID field offers direct control over bytes that appear early in packets sent by the peer; see Section 5.1. The Token field in Initial packets offers a server control over other bytes of Initial packets; see Section 17.2.2.

There are no measures in this version of QUIC to prevent indirect control over the encrypted portions of packets. It is necessary to assume that endpoints are able to control the contents of frames that a peer sends, especially those frames that convey application data, such as STREAM frames. Though this depends to some degree on details of the application protocol, some control is possible in many protocol usage contexts. As the attacker has access to packet protection keys, they are likely to be capable of predicting how a peer will encrypt future packets. Successful control over datagram content then only requires that the attacker be able to predict the packet number and placement of frames in packets with some amount of reliability.

This section assumes that limiting control over datagram content is not feasible. The focus of the mitigations in subsequent sections is on limiting the ways in which datagrams that are sent prior to address validation can be used for request forgery.

21.5.2. Request Forgery with Client Initial Packets

An attacker acting as a server can choose the IP address and port on which it advertises its availability, so Initial packets from clients are assumed to be available for use in this sort of attack. The address validation implicit in the handshake ensures that – for a new connection – a client will not send other types of packet to a destination that does not understand QUIC or is not willing to accept a QUIC connection.

Initial packet protection (Section 5.2 of [QUIC-TLS]) makes it difficult for servers to control the content of Initial packets sent by clients. A client choosing an unpredictable Destination Connection ID ensures that servers are unable to control any of the encrypted portion of Initial packets from clients.

However, the Token field is open to server control and does allow a server to use clients to mount request forgery attacks. Use of tokens provided with the NEW_TOKEN frame (Section 8.1.3) offers the only option for request forgery during connection establishment.

Clients however are not obligated to use the NEW_TOKEN frame. Request forgery attacks that rely on the Token field can be avoided if clients send an empty Token field when the server address has changed from when the NEW_TOKEN frame was received.

Clients could avoid using NEW_TOKEN if the server address changes. However, not including a Token field could adversely affect performance. Servers could rely on NEW_TOKEN to enable sending of data in excess of the three times limit on sending data; see Section 8.1. In particular, this affects cases where clients use 0-RTT to request data from servers.

Sending a Retry packet (Section 17.2.5) offers a server the option to change the Token field. After sending a Retry, the server can also control the Destination Connection ID field of subsequent Initial packets from the client. This also might allow indirect control over the encrypted content of Initial packets. However, the exchange of a Retry packet validates the server's address, thereby preventing the use of subsequent Initial packets for request forgery.

21.5.3. Request Forgery with Preferred Addresses

Servers can specify a preferred address, which clients then migrate to after confirming the handshake; see Section 9.6. The Destination Connection ID field of packets that the client sends to a preferred address can be used for request forgery.

A client **MUST NOT** send non-probing frames to a preferred address prior to validating that address; see Section 8. This greatly reduces the options that a server has to control the encrypted portion of datagrams.

This document does not offer any additional countermeasures that are specific to use of preferred addresses and can be implemented by endpoints. The generic measures described in Section 21.5.6 could be used as further mitigation.

21.5.4. Request Forgery with Spoofed Migration

Clients are able to present a spoofed source address as part of an apparent connection migration to cause a server to send datagrams to that address.

The Destination Connection ID field in any packets that a server subsequently sends to this spoofed address can be used for request forgery. A client might also be able to influence the ciphertext.

A server that only sends probing packets (Section 9.1) to an address prior to address validation provides an attacker with only limited control over the encrypted portion of datagrams. However, particularly for NAT rebinding, this can adversely affect performance. If the server sends frames carrying application data, an attacker might be able to control most of the content of datagrams.

This document does not offer specific countermeasures that can be implemented by endpoints aside from the generic measures described in Section 21.5.6. However, countermeasures for address spoofing at the network level, in particular ingress filtering [BCP38], are especially effective against attacks that use spoofing and originate from an external network.

21.5.5. Request Forgery with Version Negotiation

Clients that are able to present a spoofed source address on a packet can cause a server to send a Version Negotiation packet Section 17.2.1 to that address.

The absence of size restrictions on the connection ID fields for packets of an unknown version increases the amount of data that the client controls from the resulting datagram. The first byte of this packet is not under client control and the next four bytes are zero, but the client is able to control up to 512 bytes starting from the fifth byte.

No specific countermeasures are provided for this attack, though generic protections Section 21.5.6 could apply. In this case, ingress filtering [BCP38] is also effective.

21.5.6. Generic Request Forgery Countermeasures

The most effective defense against request forgery attacks is to modify vulnerable services to use strong authentication. However, this is not always something that is within the control of a QUIC deployment. This section outlines some other steps that QUIC endpoints could take unilaterally. These additional steps are all discretionary as, depending on circumstances, they could interfere with or prevent legitimate uses.

Services offered over loopback interfaces often lack proper authentication. Endpoints MAY prevent connection attempts or migration to a loopback address. Endpoints SHOULD NOT allow connections or migration to a loopback address if the same service was previously available at a different interface or if the address was provided by a service at a non-loopback address. Endpoints that depend on these capabilities could offer an option to disable these protections.

Similarly, endpoints could regard a change in address to link-local address [RFC4291] or an address in a private use range [RFC1918] from a global, unique-local [RFC4193], or non-private address as a potential attempt at request forgery. Endpoints could refuse to use these addresses entirely, but that carries a significant risk of interfering with legitimate uses. Endpoints SHOULD NOT refuse to use an address unless they have specific knowledge about the network indicating that sending datagrams to unvalidated addresses in a given range is not safe.

Endpoints MAY choose to reduce the risk of request forgery by not including values from NEW_TOKEN frames in Initial packets or by only sending probing frames in packets prior to completing address validation. Note that this does not prevent an attacker from using the Destination Connection ID field for an attack.

Endpoints are not expected to have specific information about the location of servers that could be vulnerable targets of a request forgery attack. However, it might be possible over time to identify specific UDP ports that are common targets of attacks or particular patterns in datagrams that are used for attacks. Endpoints MAY choose to avoid sending datagrams to these ports or not send datagrams that match these patterns prior to validating the destination address. Endpoints MAY retire connection IDs containing patterns known to be problematic without using them.

Note: Modifying endpoints to apply these protections is more efficient than deploying network-based protections, as endpoints do not need to perform any additional processing when sending to an address that has been validated.

21.6. Slowloris Attacks

The attacks commonly known as Slowloris ([SLOWLORIS]) try to keep many connections to the target endpoint open and hold them open as long as possible. These attacks can be executed against a QUIC endpoint by generating the minimum amount of activity necessary to avoid being closed for inactivity. This might involve sending small amounts of data, gradually opening flow control windows in order to

control the sender rate, or manufacturing ACK frames that simulate a high loss rate.

QUIC deployments SHOULD provide mitigations for the Slowloris attacks, such as increasing the maximum number of clients the server will allow, limiting the number of connections a single IP address is allowed to make, imposing restrictions on the minimum transfer speed a connection is allowed to have, and restricting the length of time an endpoint is allowed to stay connected.

21.7. Stream Fragmentation and Reassembly Attacks

An adversarial sender might intentionally not send portions of the stream data, causing the receiver to commit resources for the unsent data. This could cause a disproportionate receive buffer memory commitment and/or the creation of a large and inefficient data structure at the receiver.

An adversarial receiver might intentionally not acknowledge packets containing stream data in an attempt to force the sender to store the unacknowledged stream data for retransmission.

The attack on receivers is mitigated if flow control windows correspond to available memory. However, some receivers will over-commit memory and advertise flow control offsets in the aggregate that exceed actual available memory. The over-commitment strategy can lead to better performance when endpoints are well behaved, but renders endpoints vulnerable to the stream fragmentation attack.

QUIC deployments SHOULD provide mitigations against stream fragmentation attacks. Mitigations could consist of avoiding over-committing memory, limiting the size of tracking data structures, delaying reassembly of STREAM frames, implementing heuristics based on the age and duration of reassembly holes, or some combination.

21.8. Stream Commitment Attack

An adversarial endpoint can open a large number of streams, exhausting state on an endpoint. The adversarial endpoint could repeat the process on a large number of connections, in a manner similar to SYN flooding attacks in TCP.

Normally, clients will open streams sequentially, as explained in Section 2.1. However, when several streams are initiated at short intervals, loss or reordering can cause STREAM frames that open streams to be received out of sequence. On receiving a higher-numbered stream ID, a receiver is required to open all intervening streams of the same type; see Section 3.2. Thus, on a new connection, opening stream 4000000 opens 1 million and 1 client-initiated bidirectional streams.

The number of active streams is limited by the `initial_max_streams_bidi` and `initial_max_streams_uni` transport parameters as updated by any received `MAX_STREAMS` frames, as explained in Section 4.6. If chosen judiciously, these limits mitigate the effect of the stream commitment attack. However, setting the limit too low could affect performance when applications expect to open large number of streams.

21.9. Peer Denial of Service

QUIC and TLS both contain frames or messages that have legitimate uses in some contexts, but that can be abused to cause a peer to expend processing resources without having any observable impact on the state of the connection.

Messages can also be used to change and revert state in small or inconsequential ways, such as by sending small increments to flow control limits.

If processing costs are disproportionately large in comparison to bandwidth consumption or effect on state, then this could allow a malicious peer to exhaust processing capacity.

While there are legitimate uses for all messages, implementations SHOULD track cost of processing relative to progress and treat excessive quantities of any non-productive packets as indicative of an attack. Endpoints MAY respond to this condition with a connection error, or by dropping packets.

21.10. Explicit Congestion Notification Attacks

An on-path attacker could manipulate the value of ECN fields in the IP header to influence the sender's rate. [RFC3168] discusses manipulations and their effects in more detail.

A limited on-path attacker can duplicate and send packets with modified ECN fields to affect the sender's rate. If duplicate packets are discarded by a receiver, an attacker will need to race the duplicate packet against the original to be successful in this

attack. Therefore, QUIC endpoints ignore the ECN field on an IP packet unless at least one QUIC packet in that IP packet is successfully processed; see Section 13.4.

21.11. Stateless Reset Oracle

Stateless resets create a possible denial of service attack analogous to a TCP reset injection. This attack is possible if an attacker is able to cause a stateless reset token to be generated for a connection with a selected connection ID. An attacker that can cause this token to be generated can reset an active connection with the same connection ID.

If a packet can be routed to different instances that share a static key, for example by changing an IP address or port, then an attacker can cause the server to send a stateless reset. To defend against this style of denial of service, endpoints that share a static key for stateless reset (see Section 10.3.2) **MUST** be arranged so that packets with a given connection ID always arrive at an instance that has connection state, unless that connection is no longer active.

More generally, servers **MUST NOT** generate a stateless reset if a connection with the corresponding connection ID could be active on any endpoint using the same static key.

In the case of a cluster that uses dynamic load balancing, it is possible that a change in load balancer configuration could occur while an active instance retains connection state. Even if an instance retains connection state, the change in routing and resulting stateless reset will result in the connection being terminated. If there is no chance of the packet being routed to the correct instance, it is better to send a stateless reset than wait for the connection to time out. However, this is acceptable only if the routing cannot be influenced by an attacker.

21.12. Version Downgrade

This document defines QUIC Version Negotiation packets in Section 6 that can be used to negotiate the QUIC version used between two endpoints. However, this document does not specify how this negotiation will be performed between this version and subsequent future versions. In particular, Version Negotiation packets do not contain any mechanism to prevent version downgrade attacks. Future versions of QUIC that use Version Negotiation packets **MUST** define a mechanism that is robust against version downgrade attacks.

21.13. Targeted Attacks by Routing

Deployments should limit the ability of an attacker to target a new connection to a particular server instance. Ideally, routing decisions are made independently of client-selected values, including addresses. Once an instance is selected, a connection ID can be selected so that later packets are routed to the same instance.

21.14. Traffic Analysis

The length of QUIC packets can reveal information about the length of the content of those packets. The PADDING frame is provided so that endpoints have some ability to obscure the length of packet content; see Section 19.1.

Note however that defeating traffic analysis is challenging and the subject of active research. Length is not the only way that information might leak. Endpoints might also reveal sensitive information through other side channels, such as the timing of packets.

22. IANA Considerations

This document establishes several registries for the management of codepoints in QUIC. These registries operate on a common set of policies as defined in Section 22.1.

22.1. Registration Policies for QUIC Registries

All QUIC registries allow for both provisional and permanent registration of codepoints. This section documents policies that are common to these registries.

22.1.1. Provisional Registrations

Provisional registration of codepoints are intended to allow for private use and experimentation with extensions to QUIC. Provisional registrations only require the inclusion of the codepoint value and contact information. However, provisional registrations could be reclaimed and reassigned for another purpose.

Provisional registrations require Expert Review, as defined in Section 4.5 of [RFC8126]. Designated expert(s) are advised that only registrations for an excessive proportion of remaining codepoint space or the very first unassigned value (see Section 22.1.2) can be rejected.

Provisional registrations will include a date field that indicates when the registration was last updated. A request to update the date on any provisional registration can be made without review from the designated expert(s).

All QUIC registries include the following fields to support provisional registration:

Value: The assigned codepoint.

Status: "Permanent" or "Provisional".

Specification: A reference to a publicly available specification for the value.

Date: The date of last update to the registration.

Change Controller: The entity that is responsible for the definition of the registration.

Contact: Contact details for the registrant.

Notes: Supplementary notes about the registration.

Provisional registrations MAY omit the Specification and Notes fields, plus any additional fields that might be required for a permanent registration. The Date field is not required as part of requesting a registration as it is set to the date the registration is created or updated.

22.1.1.2. Selecting Codepoints

New uses of codepoints from QUIC registries SHOULD use a randomly selected codepoint that excludes both existing allocations and the first unallocated codepoint in the selected space. Requests for multiple codepoints MAY use a contiguous range. This minimizes the risk that differing semantics are attributed to the same codepoint by different implementations.

Use of the first unassigned codepoint is reserved for allocation using the Standards Action policy; see Section 4.9 of [RFC8126]. The early codepoint assignment process [EARLY-ASSIGN] can be used for these values.

For codepoints that are encoded in variable-length integers (Section 16), such as frame types, codepoints that encode to four or eight bytes (that is, values 2^{14} and above) SHOULD be used unless the usage is especially sensitive to having a longer encoding.

Applications to register codepoints in QUIC registries MAY include a requested codepoint as part of the registration. IANA MUST allocate the selected codepoint if the codepoint is unassigned and the requirements of the registration policy are met.

22.1.3. Reclaiming Provisional Codepoints

A request might be made to remove an unused provisional registration from the registry to reclaim space in a registry, or portion of the registry (such as the 64-16383 range for codepoints that use variable-length encodings). This SHOULD be done only for the codepoints with the earliest recorded date and entries that have been updated less than a year prior SHOULD NOT be reclaimed.

A request to remove a codepoint MUST be reviewed by the designated expert(s). The expert(s) MUST attempt to determine whether the codepoint is still in use. Experts are advised to contact the listed contacts for the registration, plus as wide a set of protocol implementers as possible in order to determine whether any use of the codepoint is known. The expert(s) are advised to allow at least four weeks for responses.

If any use of the codepoints is identified by this search or a request to update the registration is made, the codepoint MUST NOT be reclaimed. Instead, the date on the registration is updated. A note might be added for the registration recording relevant information that was learned.

If no use of the codepoint was identified and no request was made to update the registration, the codepoint MAY be removed from the registry.

This review and consultation process also applies to requests to change a provisional registration into a permanent registration, except that the goal is not to determine whether there is no use of the codepoint, but to determine that the registration is an accurate representation of any deployed usage.

22.1.4. Permanent Registrations

Permanent registrations in QUIC registries use the Specification Required policy ([RFC8126]), unless otherwise specified. The designated expert(s) verify that a specification exists and is readily accessible. Expert(s) are encouraged to be biased towards approving registrations unless they are abusive, frivolous, or actively harmful (not merely aesthetically displeasing, or architecturally dubious). The creation of a registry MAY specify additional constraints on permanent registrations.

The creation of a registry MAY identify a range of codepoints where registrations are governed by a different registration policy. For instance, the frame type registry in Section 22.4 has a stricter policy for codepoints in the range from 0 to 63.

Any stricter requirements for permanent registrations do not prevent provisional registrations for affected codepoints. For instance, a provisional registration for a frame type of 61 could be requested.

All registrations made by Standards Track publications MUST be permanent.

All registrations in this document are assigned a permanent status and list a change controller of the IETF and a contact of the QUIC working group (quic@ietf.org).

22.2. QUIC Versions Registry

IANA [SHALL add/has added] a registry for "QUIC Versions" under a "QUIC" heading.

The "QUIC Versions" registry governs a 32-bit space; see Section 15. This registry follows the registration policy from Section 22.1. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]).

The codepoint of 0x00000001 to the protocol is assigned with permanent status to the protocol defined in this document. The codepoint of 0x00000000 is permanently reserved; the note for this codepoint [shall] indicate[s] that this version is reserved for Version Negotiation.

All codepoints that follow the pattern 0x?a?a?a are reserved and MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

[[RFC editor: please remove the following note before publication.]]

IANA note: Several pre-standardization versions will likely be in use at the time of publication. There is no need to document these in an RFC, but recording information about these version will ensure that the information in the registry is accurate. The document editors or working group chairs can facilitate getting the necessary information.

22.3. QUIC Transport Parameter Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Parameters" under a "QUIC" heading.

The "QUIC Transport Parameters" registry governs a 62-bit space. This registry follows the registration policy from Section 22.1. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]).

In addition to the fields in Section 22.1.1, permanent registrations in this registry MUST include the following field:

Parameter Name: A short mnemonic for the parameter.

The initial contents of this registry are shown in Table 6.

Value	Parameter Name	Specification
0x00	original_destination_connection_id	Section 18.2
0x01	max_idle_timeout	Section 18.2
0x02	stateless_reset_token	Section 18.2
0x03	max_udp_payload_size	Section 18.2
0x04	initial_max_data	Section 18.2
0x05	initial_max_stream_data_bidi_local	Section 18.2
0x06	initial_max_stream_data_bidi_remote	Section 18.2
0x07	initial_max_stream_data_uni	Section 18.2
0x08	initial_max_streams_bidi	Section 18.2
0x09	initial_max_streams_uni	Section 18.2
0x0a	ack_delay_exponent	Section 18.2
0x0b	max_ack_delay	Section 18.2
0x0c	disable_active_migration	Section 18.2
0x0d	preferred_address	Section 18.2
0x0e	active_connection_id_limit	Section 18.2
0x0f	initial_source_connection_id	Section 18.2
0x10	retry_source_connection_id	Section 18.2

Table 6: Initial QUIC Transport Parameters Entries

Each value of the format " $31 * N + 27$ " for integer values of N (that is, 27, 58, 89, ...) are reserved; these values MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

22.4. QUIC Frame Types Registry

IANA [SHALL add/has added] a registry for "QUIC Frame Types" under a "QUIC" heading.

The "QUIC Frame Types" registry governs a 62-bit space. This registry follows the registration policy from Section 22.1. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Section 4.9 and 4.10 of [RFC8126].

In addition to the fields in Section 22.1.1, permanent registrations in this registry MUST include the following field:

Frame Name: A short mnemonic for the frame type.

In addition to the advice in Section 22.1, specifications for new permanent registrations SHOULD describe the means by which an endpoint might determine that it can send the identified type of frame. An accompanying transport parameter registration is expected for most registrations; see Section 22.3. Specifications for permanent registrations also need to describe the format and assigned semantics of any fields in the frame.

The initial contents of this registry are tabulated in Table 3. Note that the registry does not include the "Pkts" and "Spec" columns from Table 3.

22.5. QUIC Transport Error Codes Registry

IANA [SHALL add/has added] a registry for "QUIC Transport Error Codes" under a "QUIC" heading.

The "QUIC Transport Error Codes" registry governs a 62-bit space. This space is split into three regions that are governed by different policies. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Section 4.9 and 4.10 of [RFC8126].

In addition to the fields in Section 22.1.1, permanent registrations in this registry MUST include the following fields:

Code: A short mnemonic for the parameter.

Description: A brief description of the error code semantics, which MAY be a summary if a specification reference is provided.

The initial contents of this registry are shown in Table 7.

Value	Code	Description	Specification
0x0	NO_ERROR	No error	Section 20
0x1	INTERNAL_ERROR	Implementation error	Section 20
0x2	CONNECTION_REFUSED	Server refuses a connection	Section 20
0x3	FLOW_CONTROL_ERROR	Flow control error	Section 20
0x4	STREAM_LIMIT_ERROR	Too many streams opened	Section 20
0x5	STREAM_STATE_ERROR	Frame received in invalid stream state	Section 20
0x6	FINAL_SIZE_ERROR	Change to final size	Section 20
0x7	FRAME_ENCODING_ERROR	Frame encoding error	Section 20
0x8	TRANSPORT_PARAMETER_ERROR	Error in transport parameters	Section 20
0x9	CONNECTION_ID_LIMIT_ERROR	Too many connection IDs received	Section 20
0xa	PROTOCOL_VIOLATION	Generic protocol violation	Section 20
0xb	INVALID_TOKEN	Invalid Token Received	Section 20
0xc	APPLICATION_ERROR	Application error	Section 20
0xd	CRYPTO_BUFFER_EXCEEDED	CRYPTO data buffer overflowed	Section 20

0xe	KEY_UPDATE_ERROR	Invalid packet protection update	Section 20
0xf	AEAD_LIMIT_REACHED	Excessive use of packet protection keys	Section 20
0x10	NO_VIABLE_PATH	No viable network path exists	Section 20

Table 7: Initial QUIC Transport Error Codes Entries

23. References

23.1. Normative References

- [BCP38] Ferguson, P. and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", BCP 38, RFC 2827, DOI 10.17487/RFC2827, May 2000, <<https://www.rfc-editor.org/info/rfc2827>>.
- [DPLPMTUD] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <<https://www.rfc-editor.org/info/rfc8899>>.
- [EARLY-ASSIGN] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, DOI 10.17487/RFC7120, January 2014, <<https://www.rfc-editor.org/info/rfc7120>>.
- [IPv4] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <<https://www.rfc-editor.org/info/rfc791>>.
- [QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-invariants-13, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-invariants-13>>.

[QUIC-RECOVERY]

Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", Work in Progress, Internet-Draft, draft-ietf-quic-recovery-34, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-recovery-34>>.

[QUIC-TLS]

Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-34, 15 January 2021, <<https://tools.ietf.org/html/draft-ietf-quic-tls-34>>.

[RFC1191]

Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <<https://www.rfc-editor.org/info/rfc1191>>.

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

[RFC3168]

Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <<https://www.rfc-editor.org/info/rfc3168>>.

[RFC3629]

Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.

[RFC6437]

Amante, S., Carpenter, B., Jiang, S., and J. Rajahalme, "IPv6 Flow Label Specification", RFC 6437, DOI 10.17487/RFC6437, November 2011, <<https://www.rfc-editor.org/info/rfc6437>>.

[RFC8085]

Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.

[RFC8126]

Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.

[RFC8174]

Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

- [RFC8201] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <<https://www.rfc-editor.org/info/rfc8201>>.
- [RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <<https://www.rfc-editor.org/info/rfc8311>>.
- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [UDP] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.

23.2. Informative References

- [AEAD] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.
- [ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.
- [COOKIE] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [CSRF] Barth, A., Jackson, C., and J. Mitchell, "Robust defenses for cross-site request forgery", Proceedings of the 15th ACM conference on Computer and communications security - CCS '08, DOI 10.1145/1455770.1455782, 2008, <<https://doi.org/10.1145/1455770.1455782>>.
- [EARLY-DESIGN] Roskind, J., "QUIC: Multiplexed Transport Over UDP", 2 December 2013, <<https://goo.gl/dMVtFi>>.

- [GATEWAY] Hätönen, S., Nyrhinen, A., Eggert, L., Strowes, S., Sarolahti, P., and M. Kojo, "An experimental study of home gateway characteristics", Proceedings of the 10th annual conference on Internet measurement - IMC '10, DOI 10.1145/1879141.1879174, 2010, <<https://doi.org/10.1145/1879141.1879174>>.
- [HTTP2] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [IPv6] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <<https://www.rfc-editor.org/info/rfc8200>>.
- [QUIC-MANAGEABILITY] Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", Work in Progress, Internet-Draft, draft-ietf-quic-manageability-08, 2 November 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-manageability-08.txt>>.
- [RANDOM] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4086>>.
- [RFC1812] Baker, F., Ed., "Requirements for IP Version 4 Routers", RFC 1812, DOI 10.17487/RFC1812, June 1995, <<https://www.rfc-editor.org/info/rfc1812>>.
- [RFC1918] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., and E. Lear, "Address Allocation for Private Internets", BCP 5, RFC 1918, DOI 10.17487/RFC1918, February 1996, <<https://www.rfc-editor.org/info/rfc1918>>.
- [RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <<https://www.rfc-editor.org/info/rfc2018>>.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.

- [RFC3449] Balakrishnan, H., Padmanabhan, V., Fairhurst, G., and M. Sooriyabandara, "TCP Performance Implications of Network Path Asymmetry", BCP 69, RFC 3449, DOI 10.17487/RFC3449, December 2002, <<https://www.rfc-editor.org/info/rfc3449>>.
- [RFC4193] Hinden, R. and B. Haberman, "Unique Local IPv6 Unicast Addresses", RFC 4193, DOI 10.17487/RFC4193, October 2005, <<https://www.rfc-editor.org/info/rfc4193>>.
- [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <<https://www.rfc-editor.org/info/rfc4291>>.
- [RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <<https://www.rfc-editor.org/info/rfc4443>>.
- [RFC4787] Audet, F., Ed. and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP", BCP 127, RFC 4787, DOI 10.17487/RFC4787, January 2007, <<https://www.rfc-editor.org/info/rfc4787>>.
- [RFC4941] Narten, T., Draves, R., and S. Krishnan, "Privacy Extensions for Stateless Address Autoconfiguration in IPv6", RFC 4941, DOI 10.17487/RFC4941, September 2007, <<https://www.rfc-editor.org/info/rfc4941>>.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <<https://www.rfc-editor.org/info/rfc5681>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <<https://www.rfc-editor.org/info/rfc7983>>.
- [RFC8087] Fairhurst, G. and M. Welzl, "The Benefits of Using Explicit Congestion Notification (ECN)", RFC 8087, DOI 10.17487/RFC8087, March 2017, <<https://www.rfc-editor.org/info/rfc8087>>.

[SEC-CONS] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <<https://www.rfc-editor.org/info/rfc3552>>.

[SLOWLORIS] RSnake Hansen, R., "Welcome to Slowloris...", June 2009, <<https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>>.

Appendix A. Pseudocode

The pseudocode in this section describes sample algorithms. These algorithms are intended to be correct and clear, rather than being optimally performant.

The pseudocode segments in this section are licensed as Code Components; see the copyright notice.

A.1. Sample Variable-Length Integer Decoding

The pseudocode in Figure 45 shows how a variable-length integer can be read from a stream of bytes. The function ReadVarint takes a single argument, a sequence of bytes which can be read in network byte order.

```
ReadVarint(data):
    // The length of variable-length integers is encoded in the
    // first two bits of the first byte.
    v = data.next_byte()
    prefix = v >> 6
    length = 1 << prefix

    // Once the length is known, remove these bits and read any
    // remaining bytes.
    v = v & 0x3f
    repeat length-1 times:
        v = (v << 8) + data.next_byte()
    return v
```

Figure 45: Sample Variable-Length Integer Decoding Algorithm

For example, the eight-byte sequence 0xc2197c5eff14e88c decodes to the decimal value 151,288,809,941,952,652; the four-byte sequence 0x9d7f3e7d decodes to 494,878,333; the two-byte sequence 0x7bbd decodes to 15,293; and the single byte 0x25 decodes to 37 (as does the two-byte sequence 0x4025).

A.2. Sample Packet Number Encoding Algorithm

The pseudocode in Figure 46 shows how an implementation can select an appropriate size for packet number encodings.

The EncodePacketNumber function takes two arguments:

- * full_pn is the full packet number of the packet being sent.
- * largest_acked is the largest packet number which has been acknowledged by the peer in the current packet number space, if any.

EncodePacketNumber(full_pn, largest_acked):

```
// The number of bits must be at least one more
// than the base-2 logarithm of the number of contiguous
// unacknowledged packet numbers, including the new packet.
if largest_acked is None:
    num_unacked = full_pn + 1
else:
    num_unacked = full_pn - largest_acked

min_bits = log(num_unacked, 2) + 1
num_bytes = ceil(min_bits / 8)

// Encode the integer value and truncate to
// the num_bytes least-significant bytes.
return encode(full_pn, num_bytes)
```

Figure 46: Sample Packet Number Encoding Algorithm

For example, if an endpoint has received an acknowledgment for packet 0xab8bc and is sending a packet with a number of 0xac5c02, there are 29,519 (0x734f) outstanding packets. In order to represent at least twice this range (59,038 packets, or 0xe69e), 16 bits are required.

In the same state, sending a packet with a number of 0xace8fe uses the 24-bit encoding, because at least 18 bits are required to represent twice the range (131,182 packets, or 0x2006e).

A.3. Sample Packet Number Decoding Algorithm

The pseudocode in Figure 47 includes an example algorithm for decoding packet numbers after header protection has been removed.

The DecodePacketNumber function takes three arguments:

- * `largest_pn` is the largest packet number that has been successfully processed in the current packet number space.
- * `truncated_pn` is the value of the Packet Number field.
- * `pn_nbits` is the number of bits in the Packet Number field (8, 16, 24, or 32).

```
DecodePacketNumber(largest_pn, truncated_pn, pn_nbits):
    expected_pn = largest_pn + 1
    pn_win      = 1 << pn_nbits
    pn_hwin     = pn_win / 2
    pn_mask     = pn_win - 1
    // The incoming packet number should be greater than
    // expected_pn - pn_hwin and less than or equal to
    // expected_pn + pn_hwin
    //
    // This means we cannot just strip the trailing bits from
    // expected_pn and add the truncated_pn because that might
    // yield a value outside the window.
    //
    // The following code calculates a candidate value and
    // makes sure it's within the packet number window.
    // Note the extra checks to prevent overflow and underflow.
    candidate_pn = (expected_pn & ~pn_mask) | truncated_pn
    if candidate_pn <= expected_pn - pn_hwin and
        candidate_pn < (1 << 62) - pn_win:
        return candidate_pn + pn_win
    if candidate_pn > expected_pn + pn_hwin and
        candidate_pn >= pn_win:
        return candidate_pn - pn_win
    return candidate_pn
```

Figure 47: Sample Packet Number Decoding Algorithm

For example, if the highest successfully authenticated packet had a packet number of 0xa82f30ea, then a packet containing a 16-bit value of 0x9b32 will be decoded as 0xa82f9b32.

A.4. Sample ECN Validation Algorithm

Each time an endpoint commences sending on a new network path, it determines whether the path supports ECN; see Section 13.4. If the path supports ECN, the goal is to use ECN. Endpoints might also periodically reassess a path that was determined to not support ECN.

This section describes one method for testing new paths. This algorithm is intended to show how a path might be tested for ECN support. Endpoints can implement different methods.

The path is assigned an ECN state that is one of "testing", "unknown", "failed", or "capable". On paths with a "testing" or "capable" state the endpoint sends packets with an ECT marking, by default ECT(0); otherwise, the endpoint sends unmarked packets.

To start testing a path, the ECN state is set to "testing" and existing ECN counts are remembered as a baseline.

The testing period runs for a number of packets or a limited time, as determined by the endpoint. The goal is not to limit the duration of the testing period, but to ensure that enough marked packets are sent for received ECN counts to provide a clear indication of how the path treats marked packets. Section 13.4.2 suggests limiting this to 10 packets or 3 times the probe timeout.

After the testing period ends, the ECN state for the path becomes "unknown". From the "unknown" state, successful validation of the ECN counts an ACK frame (see Section 13.4.2.1) causes the ECN state for the path to become "capable", unless no marked packet has been acknowledged.

If validation of ECN counts fails at any time, the ECN state for the affected path becomes "failed". An endpoint can also mark the ECN state for a path as "failed" if marked packets are all declared lost or if they are all CE marked.

Following this algorithm ensures that ECN is rarely disabled for paths that properly support ECN. Any path that incorrectly modifies markings will cause ECN to be disabled. For those rare cases where marked packets are discarded by the path, the short duration of the testing period limits the number of losses incurred.

Appendix B. Change Log

RFC Editor's Note: Please remove this section prior to publication of a final version of this document.

Issue and pull request numbers are listed with a leading octothorp.

B.1. Since draft-ietf-quic-transport-32

- * Endpoints are required to limit the total data they send in response to an apparent connection migration to three times what was received (#4257, #4264)

- * Added an error code for path validation failures (#4257, #4331)
- * Defined DoS protections for clients during the handshake (#4259, #4330, #4344)
- * Prohibited connection errors when datagrams are not the required size (#4273, #4342)
- * Stop using initial timeout for path validation (#4261, #4262, #4263).
- * A number of improvements to IANA considerations:
 - Added a registry for versions (#4345, #4280)
 - Clarified rules for first reserved value (#4378, #4389)
 - Reserved values are not added to the registry (#4372, #4428)
- * Added final version numbers (#4430)

B.2. Since draft-ietf-quic-transport-31

- * Require expansion of datagrams to ensure that a path supports at least 1200 bytes in both directions:
 - During the handshake ack-eliciting Initial packets from the server need to be expanded (#4183, #4188)
 - Path validation now requires packets containing PATH_CHALLENGE and PATH_RESPONSE to be expanded and PATH_RESPONSE is sent on the same network path (#4216, #4226)
- * Though senders need to expand datagrams in some cases, receivers cannot enforce this requirement (#4253, #4254)
- * Split contact into contact and change controller for IANA registrations (#4230, #4239)

B.3. Since draft-ietf-quic-transport-30

- * Use TRANSPORT_PARAMETER_ERROR for an invalid transport parameter (#4099, #4100)
- * Add a new error code for AEAD_LIMIT_REACHED code to avoid conflict (#4087, #4088)

- * Allow use of address validation token when server address changes (#4076, #4089)

B.4. Since draft-ietf-quic-transport-29

- * Require the same connection ID on coalesced packets (#3800, #3930)
- * Allow caching of packets that can't be decrypted, by allowing the reported acknowledgment delay to exceed `max_ack_delay` prior to confirming the handshake (#3821, #3980, #4035, #3874)
- * Allow connection ID to be used for address validation (#3834, #3924)
- * Required protocol operations are no longer directed at implementations, but are features provided to application protocols (#3838, #3935)
- * Narrow requirements for reset of congestion state on path change (#3842, #3945)
- * Add a three times amplification limit for sending of `CONNECTION_CLOSE` with reduced state (#3845, #3864)
- * Change error code for invalid `RETIRE_CONNECTION_ID` frames (#3860, #3861)
- * Recommend retention of state for lost packets to allow for late arrival and avoid unnecessary retransmission (#3956, #3957)
- * Allow a server to reject connections if a client reuses packet numbers after Retry (#3989, #3990)
- * Limit recommendation for immediate acknowledgment to when ack-eliciting packets are reordered (#4001, #4000)

B.5. Since draft-ietf-quic-transport-28

- * Made `SERVER_BUSY` error (0x2) more generic, now `CONNECTION_REFUSED` (#3709, #3690, #3694)
- * Allow `TRANSPORT_PARAMETER_ERROR` when validating connection IDs (#3703, #3691)
- * Integrate QUIC-specific language from draft-ietf-tsvwg-datagram-plpmtud (#3695, #3702)

- * `disable_active_migration` does not apply to the addresses offered in `server_preferred_address` (#3608, #3670)

B.6. Since draft-ietf-quic-transport-27

- * Allowed `CONNECTION_CLOSE` in any packet number space, with a requirement to use a new transport-level error for application-specific errors in Initial and Handshake packets (#3430, #3435, #3440)
- * Clearer requirements for address validation (#2125, #3327)
- * Security analysis of handshake and migration (#2143, #2387, #2925)
- * The entire payload of a datagram is used when counting bytes for mitigating amplification attacks (#3333, #3470)
- * Connection IDs can be used at any time, including in the handshake (#3348, #3560, #3438, #3565)
- * Only one ACK should be sent for each instance of reordering (#3357, #3361)
- * Remove text allowing a server to proceed with a bad Retry token (#3396, #3398)
- * Ignore `active_connection_id_limit` with a zero-length connection ID (#3427, #3426)
- * Require `active_connection_id_limit` be remembered for 0-RTT (#3423, #3425)
- * Require `ack_delay` not be remembered for 0-RTT (#3433, #3545)
- * Redefined `max_packet_size` to `max_udp_datagram_size` (#3471, #3473)
- * Guidance on limiting outstanding attempts to retire connection IDs (#3489, #3509, #3557, #3547)
- * Restored text on dropping bogus Version Negotiation packets (#3532, #3533)
- * Clarified that largest acknowledged needs to be saved, but not necessarily signaled in all cases (#3541, #3581)
- * Addressed linkability risk with the use of `preferred_address` (#3559, #3563)

- * Added authentication of handshake connection IDs (#3439, #3499)
 - * Opening a stream in the wrong direction is an error (#3527)
- B.7. Since draft-ietf-quic-transport-26
- * Change format of transport parameters to use varints (#3294, #3169)
- B.8. Since draft-ietf-quic-transport-25
- * Define the use of CONNECTION_CLOSE prior to establishing connection state (#3269, #3297, #3292)
 - * Allow use of address validation tokens after client address changes (#3307, #3308)
 - * Define the timer for address validation (#2910, #3339)
- B.9. Since draft-ietf-quic-transport-24
- * Added HANDSHAKE_DONE to signal handshake confirmation (#2863, #3142, #3145)
 - * Add integrity check to Retry packets (#3014, #3274, #3120)
 - * Specify handling of reordered NEW_CONNECTION_ID frames (#3194, #3202)
 - * Require checking of sequence numbers in RETIRE_CONNECTION_ID (#3037, #3036)
 - * active_connection_id_limit is enforced (#3193, #3197, #3200, #3201)
 - * Correct overflow in packet number decode algorithm (#3187, #3188)
 - * Allow use of CRYPTO_BUFFER_EXCEEDED for CRYPTO frame errors (#3258, #3186)
 - * Define applicability and scope of NEW_TOKEN (#3150, #3152, #3155, #3156)
 - * Tokens from Retry and NEW_TOKEN must be differentiated (#3127, #3128)
 - * Allow CONNECTION_CLOSE in response to invalid token (#3168, #3107)

- * Treat an invalid CONNECTION_CLOSE as an invalid frame (#2475, #3230, #3231)
- * Throttle when sending CONNECTION_CLOSE after discarding state (#3095, #3157)
- * Application-variant of CONNECTION_CLOSE can only be sent in 0-RTT or 1-RTT packets (#3158, #3164)
- * Advise sending while blocked to avoid idle timeout (#2744, #3266)
- * Define error codes for invalid frames (#3027, #3042)
- * Idle timeout is symmetric (#2602, #3099)
- * Prohibit IP fragmentation (#3243, #3280)
- * Define the use of provisional registration for all registries (#3109, #3020, #3102, #3170)
- * Packets on one path must not adjust values for a different path (#2909, #3139)

B.10. Since draft-ietf-quic-transport-23

- * Allow ClientHello to span multiple packets (#2928, #3045)
- * Client Initial size constraints apply to UDP datagram payload (#3053, #3051)
- * Stateless reset changes (#2152, #2993)
 - tokens need to be compared in constant time
 - detection uses UDP datagrams, not packets
 - tokens cannot be reused (#2785, #2968)
- * Clearer rules for sharing of UDP ports and use of connection IDs when doing so (#2844, #2851)
- * A new connection ID is necessary when responding to migration (#2778, #2969)
- * Stronger requirements for connection ID retirement (#3046, #3096)
- * NEW_TOKEN cannot be empty (#2978, #2977)

- * PING can be sent at any encryption level (#3034, #3035)
- * CONNECTION_CLOSE is not ack-eliciting (#3097, #3098)
- * Frame encoding error conditions updated (#3027, #3042)
- * Non-ack-eliciting packets cannot be sent in response to non-ack-eliciting packets (#3100, #3104)
- * Servers have to change connection IDs in Retry (#2837, #3147)

B.11. Since draft-ietf-quic-transport-22

- * Rules for preventing correlation by connection ID tightened (#2084, #2929)
- * Clarified use of CONNECTION_CLOSE in Handshake packets (#2151, #2541, #2688)
- * Discourage regressions of largest acknowledged in ACK (#2205, #2752)
- * Improved robustness of validation process for ECN counts (#2534, #2752)
- * Require endpoints to ignore spurious migration attempts (#2342, #2893)
- * Transport parameter for disabling migration clarified to allow NAT rebinding (#2389, #2893)
- * Document principles for defining new error codes (#2388, #2880)
- * Reserve transport parameters for greasing (#2550, #2873)
- * A maximum ACK delay of 0 is used for handshake packet number spaces (#2646, #2638)
- * Improved rules for use of congestion control state on new paths (#2685, #2918)
- * Removed recommendation to coordinate spin for multiple connections that share a path (#2763, #2882)
- * Allow smaller stateless resets and recommend a smaller minimum on packets that might trigger a stateless reset (#2770, #2869, #2927, #3007).

- * Provide guidance around the interface to QUIC as used by application protocols (#2805, #2857)
- * Frames other than STREAM can cause STREAM_LIMIT_ERROR (#2825, #2826)
- * Tighter rules about processing of rejected 0-RTT packets (#2829, #2840, #2841)
- * Explanation of the effect of Retry on 0-RTT packets (#2842, #2852)
- * Cryptographic handshake needs to provide server transport parameter encryption (#2920, #2921)
- * Moved ACK generation guidance from recovery draft to transport draft (#1860, #2916).

B.12. Since draft-ietf-quic-transport-21

- * Connection ID lengths are now one octet, but limited in version 1 to 20 octets of length (#2736, #2749)

B.13. Since draft-ietf-quic-transport-20

- * Error codes are encoded as variable-length integers (#2672, #2680)
- * NEW_CONNECTION_ID includes a request to retire old connection IDs (#2645, #2769)
- * Tighter rules for generating and explicitly eliciting ACK frames (#2546, #2794)
- * Recommend having only one packet per encryption level in a datagram (#2308, #2747)
- * More normative language about use of stateless reset (#2471, #2574)
- * Allow reuse of stateless reset tokens (#2732, #2733)
- * Allow, but not require, enforcing non-duplicate transport parameters (#2689, #2691)
- * Added an active_connection_id_limit transport parameter (#1994, #1998)
- * max_ack_delay transport parameter defaults to 0 (#2638, #2646)

- * When sending 0-RTT, only remembered transport parameters apply (#2458, #2360, #2466, #2461)
- * Define handshake completion and confirmation; define clearer rules when it encryption keys should be discarded (#2214, #2267, #2673)
- * Prohibit path migration prior to handshake confirmation (#2309, #2370)
- * PATH_RESPONSE no longer needs to be received on the validated path (#2582, #2580, #2579, #2637)
- * PATH_RESPONSE frames are not stored and retransmitted (#2724, #2729)
- * Document hack for enabling routing of ICMP when doing PMTU probing (#1243, #2402)

B.14. Since draft-ietf-quic-transport-19

- * Refine discussion of 0-RTT transport parameters (#2467, #2464)
- * Fewer transport parameters need to be remembered for 0-RTT (#2624, #2467)
- * Spin bit text incorporated (#2564)
- * Close the connection when maximum stream ID in MAX_STREAMS exceeds $2^{62} - 1$ (#2499, #2487)
- * New connection ID required for intentional migration (#2414, #2413)
- * Connection ID issuance can be rate-limited (#2436, #2428)
- * The "QUIC bit" is ignored in Version Negotiation (#2400, #2561)
- * Initial packets from clients need to be padded to 1200 unless a Handshake packet is sent as well (#2522, #2523)
- * CRYPTO frames can be discarded if too much data is buffered (#1834, #2524)
- * Stateless reset uses a short header packet (#2599, #2600)

B.15. Since draft-ietf-quic-transport-18

- * Removed version negotiation; version negotiation, including authentication of the result, will be addressed in the next version of QUIC (#1773, #2313)
- * Added discussion of the use of IPv6 flow labels (#2348, #2399)
- * A connection ID can't be retired in a packet that uses that connection ID (#2101, #2420)
- * Idle timeout transport parameter is in milliseconds (from seconds) (#2453, #2454)
- * Endpoints are required to use new connection IDs when they use new network paths (#2413, #2414)
- * Increased the set of permissible frames in 0-RTT (#2344, #2355)

B.16. Since draft-ietf-quic-transport-17

- * Stream-related errors now use STREAM_STATE_ERROR (#2305)
- * Endpoints discard initial keys as soon as handshake keys are available (#1951, #2045)
- * Expanded conditions for ignoring ICMP packet too big messages (#2108, #2161)
- * Remove rate control from PATH_CHALLENGE/PATH_RESPONSE (#2129, #2241)
- * Endpoints are permitted to discard malformed initial packets (#2141)
- * Clarified ECN implementation and usage requirements (#2156, #2201)
- * Disable ECN count verification for packets that arrive out of order (#2198, #2215)
- * Use Probe Timeout (PTO) instead of RTO (#2206, #2238)
- * Loosen constraints on retransmission of ACK ranges (#2199, #2245)
- * Limit Retry and Version Negotiation to once per datagram (#2259, #2303)
- * Set a maximum value for max_ack_delay transport parameter (#2282, #2301)

- * Allow server preferred address for both IPv4 and IPv6 (#2122, #2296)
- * Corrected requirements for migration to a preferred address (#2146, #2349)
- * ACK of non-existent packet is illegal (#2298, #2302)

B.17. Since draft-ietf-quic-transport-16

- * Stream limits are defined as counts, not maximums (#1850, #1906)
- * Require amplification attack defense after closing (#1905, #1911)
- * Remove reservation of application error code 0 for STOPPING (#1804, #1922)
- * Renumbered frames (#1945)
- * Renumbered transport parameters (#1946)
- * Numeric transport parameters are expressed as varints (#1608, #1947, #1955)
- * Reorder the NEW_CONNECTION_ID frame (#1952, #1963)
- * Rework the first byte (#2006)
 - Fix the 0x40 bit
 - Change type values for long header
 - Add spin bit to short header (#631, #1988)
 - Encrypt the remainder of the first byte (#1322)
 - Move packet number length to first byte
 - Move ODCIL to first byte of retry packets
 - Simplify packet number protection (#1575)
- * Allow STOP_SENDING to open a remote bidirectional stream (#1797, #2013)
- * Added mitigation for off-path migration attacks (#1278, #1749, #2033)

- * Don't let the PMTU to drop below 1280 (#2063, #2069)
- * Require peers to replace retired connection IDs (#2085)
- * Servers are required to ignore Version Negotiation packets (#2088)
- * Tokens are repeated in all Initial packets (#2089)
- * Clarified how PING frames are sent after loss (#2094)
- * Initial keys are discarded once Handshake are available (#1951, #2045)
- * ICMP PTB validation clarifications (#2161, #2109, #2108)

B.18. Since draft-ietf-quic-transport-15

Substantial editorial reorganization; no technical changes.

B.19. Since draft-ietf-quic-transport-14

- * Merge ACK and ACK_ECN (#1778, #1801)
- * Explicitly communicate max_ack_delay (#981, #1781)
- * Validate original connection ID after Retry packets (#1710, #1486, #1793)
- * Idle timeout is optional and has no specified maximum (#1765)
- * Update connection ID handling; add RETIRE_CONNECTION_ID type (#1464, #1468, #1483, #1484, #1486, #1495, #1729, #1742, #1799, #1821)
- * Include a Token in all Initial packets (#1649, #1794)
- * Prevent handshake deadlock (#1764, #1824)

B.20. Since draft-ietf-quic-transport-13

- * Streams open when higher-numbered streams of the same type open (#1342, #1549)
- * Split initial stream flow control limit into 3 transport parameters (#1016, #1542)
- * All flow control transport parameters are optional (#1610)

- * Removed UNSOLICITED_PATH_RESPONSE error code (#1265, #1539)
- * Permit stateless reset in response to any packet (#1348, #1553)
- * Recommended defense against stateless reset spoofing (#1386, #1554)
- * Prevent infinite stateless reset exchanges (#1443, #1627)
- * Forbid processing of the same packet number twice (#1405, #1624)
- * Added a packet number decoding example (#1493)
- * More precisely define idle timeout (#1429, #1614, #1652)
- * Corrected format of Retry packet and prevented looping (#1492, #1451, #1448, #1498)
- * Permit 0-RTT after receiving Version Negotiation or Retry (#1507, #1514, #1621)
- * Permit Retry in response to 0-RTT (#1547, #1552)
- * Looser verification of ECN counters to account for ACK loss (#1555, #1481, #1565)
- * Remove frame type field from APPLICATION_CLOSE (#1508, #1528)

B.21. Since draft-ietf-quic-transport-12

- * Changes to integration of the TLS handshake (#829, #1018, #1094, #1165, #1190, #1233, #1242, #1252, #1450, #1458)
 - The cryptographic handshake uses CRYPTO frames, not stream 0
 - QUIC packet protection is used in place of TLS record protection
 - Separate QUIC packet number spaces are used for the handshake
 - Changed Retry to be independent of the cryptographic handshake
 - Added NEW_TOKEN frame and Token fields to Initial packet
 - Limit the use of HelloRetryRequest to address TLS needs (like key shares)

- * Enable server to transition connections to a preferred address (#560, #1251, #1373)
- * Added ECN feedback mechanisms and handling; new ACK_ECN frame (#804, #805, #1372)
- * Changed rules and recommendations for use of new connection IDs (#1258, #1264, #1276, #1280, #1419, #1452, #1453, #1465)
- * Added a transport parameter to disable intentional connection migration (#1271, #1447)
- * Packets from different connection ID can't be coalesced (#1287, #1423)
- * Fixed sampling method for packet number encryption; the length field in long headers includes the packet number field in addition to the packet payload (#1387, #1389)
- * Stateless Reset is now symmetric and subject to size constraints (#466, #1346)
- * Added frame type extension mechanism (#58, #1473)

B.22. Since draft-ietf-quic-transport-11

- * Enable server to transition connections to a preferred address (#560, #1251)
- * Packet numbers are encrypted (#1174, #1043, #1048, #1034, #850, #990, #734, #1317, #1267, #1079)
- * Packet numbers use a variable-length encoding (#989, #1334)
- * STREAM frames can now be empty (#1350)

B.23. Since draft-ietf-quic-transport-10

- * Swap payload length and packed number fields in long header (#1294)
- * Clarified that CONNECTION_CLOSE is allowed in Handshake packet (#1274)
- * Spin bit reserved (#1283)
- * Coalescing multiple QUIC packets in a UDP datagram (#1262, #1285)

- * A more complete connection migration (#1249)
- * Refine opportunistic ACK defense text (#305, #1030, #1185)
- * A Stateless Reset Token isn't mandatory (#818, #1191)
- * Removed implicit stream opening (#896, #1193)
- * An empty STREAM frame can be used to open a stream without sending data (#901, #1194)
- * Define stream counts in transport parameters rather than a maximum stream ID (#1023, #1065)
- * STOP_SENDING is now prohibited before streams are used (#1050)
- * Recommend including ACK in Retry packets and allow PADDING (#1067, #882)
- * Endpoints now become closing after an idle timeout (#1178, #1179)
- * Remove implication that Version Negotiation is sent when a packet of the wrong version is received (#1197)

B.24. Since draft-ietf-quic-transport-09

- * Added PATH_CHALLENGE and PATH_RESPONSE frames to replace PING with Data and PONG frame. Changed ACK frame type from 0x0e to 0x0d. (#1091, #725, #1086)
- * A server can now only send 3 packets without validating the client address (#38, #1090)
- * Delivery order of stream data is no longer strongly specified (#252, #1070)
- * Rework of packet handling and version negotiation (#1038)
- * Stream 0 is now exempt from flow control until the handshake completes (#1074, #725, #825, #1082)
- * Improved retransmission rules for all frame types: information is retransmitted, not packets or frames (#463, #765, #1095, #1053)
- * Added an error code for server busy signals (#1137)

- * Endpoints now set the connection ID that their peer uses. Connection IDs are variable length. Removed the omit_connection_id transport parameter and the corresponding short header flag. (#1089, #1052, #1146, #821, #745, #821, #1166, #1151)

B.25. Since draft-ietf-quic-transport-08

- * Clarified requirements for BLOCKED usage (#65, #924)
- * BLOCKED frame now includes reason for blocking (#452, #924, #927, #928)
- * GAP limitation in ACK Frame (#613)
- * Improved PMTUD description (#614, #1036)
- * Clarified stream state machine (#634, #662, #743, #894)
- * Reserved versions don't need to be generated deterministically (#831, #931)
- * You don't always need the draining period (#871)
- * Stateless reset clarified as version-specific (#930, #986)
- * initial_max_stream_id_x transport parameters are optional (#970, #971)
- * ACK delay assumes a default value during the handshake (#1007, #1009)
- * Removed transport parameters from NewSessionTicket (#1015)

B.26. Since draft-ietf-quic-transport-07

- * The long header now has version before packet number (#926, #939)
- * Rename and consolidate packet types (#846, #822, #847)
- * Packet types are assigned new codepoints and the Connection ID Flag is inverted (#426, #956)
- * Removed type for Version Negotiation and use Version 0 (#963, #968)
- * Streams are split into unidirectional and bidirectional (#643, #656, #720, #872, #175, #885)

- Stream limits now have separate uni- and bi-directional transport parameters (#909, #958)
- Stream limit transport parameters are now optional and default to 0 (#970, #971)
- * The stream state machine has been split into read and write (#634, #894)
- * Employ variable-length integer encodings throughout (#595)
- * Improvements to connection close
 - Added distinct closing and draining states (#899, #871)
 - Draining period can terminate early (#869, #870)
 - Clarifications about stateless reset (#889, #890)
- * Address validation for connection migration (#161, #732, #878)
- * Clearly defined retransmission rules for BLOCKED (#452, #65, #924)
- * negotiated_version is sent in server transport parameters (#710, #959)
- * Increased the range over which packet numbers are randomized (#864, #850, #964)

B.27. Since draft-ietf-quic-transport-06

- * Replaced FNV-1a with AES-GCM for all "Cleartext" packets (#554)
- * Split error code space between application and transport (#485)
- * Stateless reset token moved to end (#820)
- * 1-RTT-protected long header types removed (#848)
- * No acknowledgments during draining period (#852)
- * Remove "application close" as a separate close type (#854)
- * Remove timestamps from the ACK frame (#841)
- * Require transport parameters to only appear once (#792)

B.28. Since draft-ietf-quic-transport-05

- * Stateless token is server-only (#726)
- * Refactor section on connection termination (#733, #748, #328, #177)
- * Limit size of Version Negotiation packet (#585)
- * Clarify when and what to ack (#736)
- * Renamed STREAM_ID_NEEDED to STREAM_ID_BLOCKED
- * Clarify Keep-alive requirements (#729)

B.29. Since draft-ietf-quic-transport-04

- * Introduce STOP_SENDING frame, RESET_STREAM only resets in one direction (#165)
- * Removed GOAWAY; application protocols are responsible for graceful shutdown (#696)
- * Reduced the number of error codes (#96, #177, #184, #211)
- * Version validation fields can't move or change (#121)
- * Removed versions from the transport parameters in a NewSessionTicket message (#547)
- * Clarify the meaning of "bytes in flight" (#550)
- * Public reset is now stateless reset and not visible to the path (#215)
- * Reordered bits and fields in STREAM frame (#620)
- * Clarifications to the stream state machine (#572, #571)
- * Increased the maximum length of the Largest Acknowledged field in ACK frames to 64 bits (#629)
- * truncate_connection_id is renamed to omit_connection_id (#659)
- * CONNECTION_CLOSE terminates the connection like TCP RST (#330, #328)
- * Update labels used in HKDF-Expand-Label to match TLS 1.3 (#642)

B.30. Since draft-ietf-quic-transport-03

- * Change STREAM and RESET_STREAM layout
- * Add MAX_STREAM_ID settings

B.31. Since draft-ietf-quic-transport-02

- * The size of the initial packet payload has a fixed minimum (#267, #472)
- * Define when Version Negotiation packets are ignored (#284, #294, #241, #143, #474)
- * The 64-bit FNV-1a algorithm is used for integrity protection of unprotected packets (#167, #480, #481, #517)
- * Rework initial packet types to change how the connection ID is chosen (#482, #442, #493)
- * No timestamps are forbidden in unprotected packets (#542, #429)
- * Cryptographic handshake is now on stream 0 (#456)
- * Remove congestion control exemption for cryptographic handshake (#248, #476)
- * Version 1 of QUIC uses TLS; a new version is needed to use a different handshake protocol (#516)
- * STREAM frames have a reduced number of offset lengths (#543, #430)
- * Split some frames into separate connection- and stream- level frames (#443)
 - WINDOW_UPDATE split into MAX_DATA and MAX_STREAM_DATA (#450)
 - BLOCKED split to match WINDOW_UPDATE split (#454)
 - Define STREAM_ID_NEEDED frame (#455)
- * A NEW_CONNECTION_ID frame supports connection migration without linkability (#232, #491, #496)
- * Transport parameters for 0-RTT are retained from a previous connection (#405, #513, #512)

- A client in 0-RTT no longer required to reset excess streams (#425, #479)

- * Expanded security considerations (#440, #444, #445, #448)

B.32. Since draft-ietf-quic-transport-01

- * Defined short and long packet headers (#40, #148, #361)
- * Defined a versioning scheme and stable fields (#51, #361)
- * Define reserved version values for "greasing" negotiation (#112, #278)
- * The initial packet number is randomized (#35, #283)
- * Narrow the packet number encoding range requirement (#67, #286, #299, #323, #356)
- * Defined client address validation (#52, #118, #120, #275)
- * Define transport parameters as a TLS extension (#49, #122)
- * SCUP and COPT parameters are no longer valid (#116, #117)
- * Transport parameters for 0-RTT are either remembered from before, or assume default values (#126)
- * The server chooses connection IDs in its final flight (#119, #349, #361)
- * The server echoes the Connection ID and packet number fields when sending a Version Negotiation packet (#133, #295, #244)
- * Defined a minimum packet size for the initial handshake packet from the client (#69, #136, #139, #164)
- * Path MTU Discovery (#64, #106)
- * The initial handshake packet from the client needs to fit in a single packet (#338)
- * Forbid acknowledgment of packets containing only ACK and PADDING (#291)
- * Require that frames are processed when packets are acknowledged (#381, #341)

- * Removed the STOP_WAITING frame (#66)
- * Don't require retransmission of old timestamps for lost ACK frames (#308)
- * Clarified that frames are not retransmitted, but the information in them can be (#157, #298)
- * Error handling definitions (#335)
- * Split error codes into four sections (#74)
- * Forbid the use of Public Reset where CONNECTION_CLOSE is possible (#289)
- * Define packet protection rules (#336)
- * Require that stream be entirely delivered or reset, including acknowledgment of all STREAM frames or the RESET_STREAM, before it closes (#381)
- * Remove stream reservation from state machine (#174, #280)
- * Only stream 1 does not contribute to connection-level flow control (#204)
- * Stream 1 counts towards the maximum concurrent stream limit (#201, #282)
- * Remove connection-level flow control exclusion for some streams (except 1) (#246)
- * RESET_STREAM affects connection-level flow control (#162, #163)
- * Flow control accounting uses the maximum data offset on each stream, rather than bytes received (#378)
- * Moved length-determining fields to the start of STREAM and ACK (#168, #277)
- * Added the ability to pad between frames (#158, #276)
- * Remove error code and reason phrase from GOAWAY (#352, #355)
- * GOAWAY includes a final stream number for both directions (#347)
- * Error codes for RESET_STREAM and CONNECTION_CLOSE are now at a consistent offset (#249)

- * Defined priority as the responsibility of the application protocol (#104, #303)

B.33. Since draft-ietf-quic-transport-00

- * Replaced DIVERSIFICATION_NONCE flag with KEY_PHASE flag
- * Defined versioning
- * Reworked description of packet and frame layout
- * Error code space is divided into regions for each component
- * Use big endian for all numeric values

B.34. Since draft-hamilton-quic-transport-protocol-01

- * Adopted as base for draft-ietf-quic-tls
- * Updated authors/editors list
- * Added IANA Considerations section
- * Moved Contributors and Acknowledgments to appendices

Contributors

The original design and rationale behind this protocol draw significantly from work by Jim Roskind [EARLY-DESIGN].

The IETF QUIC Working Group received an enormous amount of support from many people. The following people provided substantive contributions to this document:

- * Alessandro Ghedini
- * Alyssa Wilk
- * Antoine Delignat-Lavaud
- * Brian Trammell
- * Christian Huitema
- * Colin Perkins
- * David Schinazi

- * Dmitri Tikhonov
- * Eric Kinnear
- * Eric Rescorla
- * Gorrry Fairhurst
- * Ian Swett
- * Igor Lubashev
- * (Kazuho Oku)
- * Lars Eggert
- * Lucas Pardue
- * Magnus Westerlund
- * Marten Seemann
- * Martin Duke
- * Mike Bishop
- * Mikkel Fahnøe Jørgensen
- * Mirja Kühlewind
- * Nick Banks
- * Nick Harper
- * Patrick McManus
- * Roberto Peon
- * Ryan Hamilton
- * Subodh Iyengar
- * Tatsuhiro Tsujikawa
- * Ted Hardie
- * Tom Jones

* Victor Vasiliev

Authors' Addresses

Jana Iyengar (editor)
Fastly

Email: jri.ietf@gmail.com

Martin Thomson (editor)
Mozilla

Email: mt@lowentropy.net

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 25, 2017

I. Johansson
Ericsson AB
February 21, 2017

ECN support in QUIC
draft-johansson-quic-ecn-01

Abstract

This memo outlines the ECN support in QUIC. The intention is that most of the material ends up updating other new or existing QUIC protocol specifications, thus it may be possible that this draft does not warrant a working group status.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 25, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Elements of ECN support	2
2.1. ECN negotiation	3
2.2. ECN bits in the IP header, semantics	4
2.3. ECN echo	4
2.4. Fallback in case of ECN fault	8
2.5. OS socket specifics, access to the ECN bits	9
2.6. Monitoring	9
3. IANA Considerations	10
4. Open questions	10
5. Security Considerations	10
6. Acknowledgements	10
7. References	11
7.1. Normative References	11
7.2. Informative References	11
Author's Address	12

1. Introduction

ECN support in transport protocols is a fundamental feature that should be included in the QUIC specification as a mandatory element. The benefits of ECN is described in [I-D.ietf-aqm-ecn-benefits]. The ECN support should be implemented to support both present and future ECN, the latter is outlined in [I-D.ietf-tsvwg-ecn-experimentation], of particular interest is the ability to discriminate between classic ECN and L4S ECN by means of differentiation between the use of the ECT(0) and ECT(1) code points. This draft does however not delve into the details of the congestion control implementation.

2. Elements of ECN support

This draft covers the following aspects of ECN support:

- o ECN negotiation
- o ECN echo
- o ECN bits in the IP header, semantics
- o Fallback in case of ECN fault

- o OS socket specifics, access to the ECN bits
- o Monitoring

2.1. ECN negotiaition

ECN support in QUIC needs to be negotiated. The reasons is that network elements may not support ECN and may either clear the ECN bits or simply discard packets that have the ECN bits set. In addition, a QUIC implementation may not have access to the ECN bits in the IP header due to OS dependent restrictions, investigations (Piers O'Hanlon) have indicated that this is in certain cases an asymmetric property, for instance while it is possible to set the ECN bits it is not possible to read them.

It is also required that the ECN negotiation does not interfere with the connection setup, in other words a failed ECN negotiation should not cause an extra roundtrip for the connection setup.

The suggested method in this draft is to add an ECN negotiation frame that is transmitted when connection setup is completed. Both peers MUST transmit the ECN negotiation frame. The ECN negotiation frame is shown below.

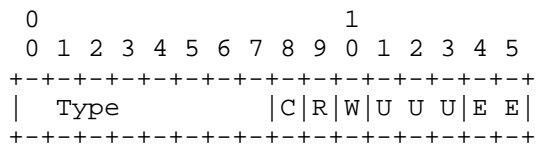


Figure 1: ECN negotiation frame

The 2nd byte contains the flags:

- o C: Challenge bit, indicates that the transmitted ECN negotiation frame is a challenge, if bit is not set then it is a response.
- o R: Possible to read ECN bits in IP header
- o W: Possible to write ECN bits in IP header
- o EE : Echo of ECN bits
- o U: Unused

A peer transmits the ECN negotiation frame with the R,W and EE bits in the 2nd byte set to '0' and the C bit set to '1'. This frame is echoed back with the flags set according to the degree of ECN support

and with the ECN bits in the IP header of the received ECN negotiation frame copied to the EE field, the C bit is '0'. As both peers MUST transmit an ECN negotiation frame there will be a total of 4 ECN negotiation frames transmitted, two challenges and two responses.

The IP header for the ECN negotiation frame should set the ECN bits to CE '11'. When the corresponding response is received then an EE pattern of '11' indicates that ECN is likely supported in the network. This does not give a full guarantee that ECN is supported in the network. Monitoring of the ECN field in the ACK-frame serves to give further indication of ECN support once ECN is turned on.

A peer is not allowed to set ECT on outgoing data packets until a ECN negotiation response that indicates that ECN is supported is received. In other words it is only the ECN negotiation frame that is allowed to set the ECN bits in the IP header.

A lack of an ECN negotiation response may indicate that the ECN challenge frame or the ECN response frame was lost or that a node in the network deliberately discards ECN-CE marked packets. The peer can transmit additional ECN challenges with given time intervals to rule out accidental packet loss. The detailed timing for this is T.B.D.

The mode mechanism in [RFC6679] can serve as input to a solution for the support of ECN in the case that OS ECN support is asymmetric. It is however unclear how a QUIC implementation can determine asymmetric ECN support in the underlying OS. For instance the method to send ECN marked packets to the local host to determine OS support does not reveal if the OS ECN support is asymmetric.

2.2. ECN bits in the IP header, semantics

The ECN bits in the IP header should be set according to the recommendations in [I-D.ietf-tsvwg-ecn-experimentation]. This means that the meaning of ECT(0) and ECT(1) differ.

2.3. ECN echo

The ECN echo should preferably go into the ACK frame [I-D.ietf-quic-transport], this is beneficial as the ECN information can then use some of the already existing data in the ACK frame for improved efficiency, this applies especially to alternatives 1 and 2 below. It is suggested that the 'U' bit in the ACK frame type is renamed 'E' to indicate the presence of an ECN field in the ACK frame, this makes it possible to omit the ECN information for the cases where ECN is not supported for the connection.

Currently there are three alternatives how to add ECN support to the ACK frames .

The first alternative inserts a one octet field that contains a 2 bit ECN echo, followed by the ACK block length. The ACK block length then dictates the number of received contiguous frames with the indicated ECN echo.

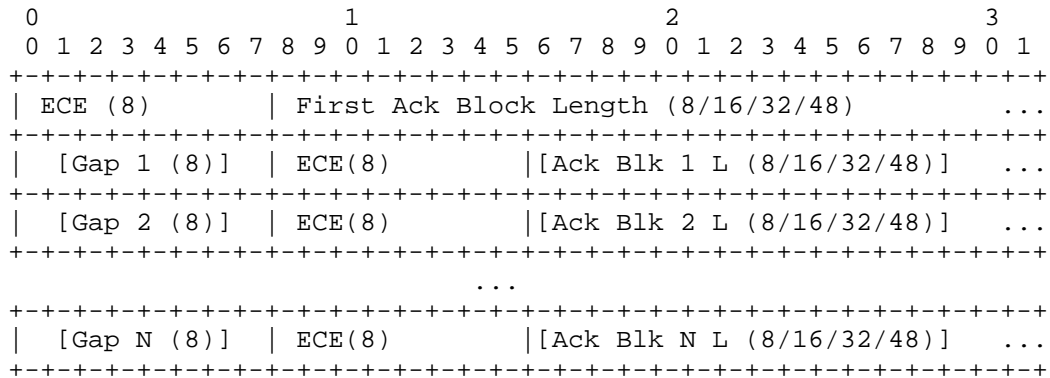


Figure 2: ECN field in ACK frame ACK block, alt 1

The second alternative encodes a variable length field that contains the ECN echoes for the frames listed in the ACK blocks. The length of the field is inferred from the ACK block lengths. No ECN echoes are indicated for the gaps (it is, after all, impossible to indicate status of the ECN bits for lost packets). For instance if the ACK blocks list 10 frames, then the length of the ECN echo field becomes $2 \times 10 = 20$ bits, with additional 4 bits of padding the ECN echo field will then become 3 octets long.

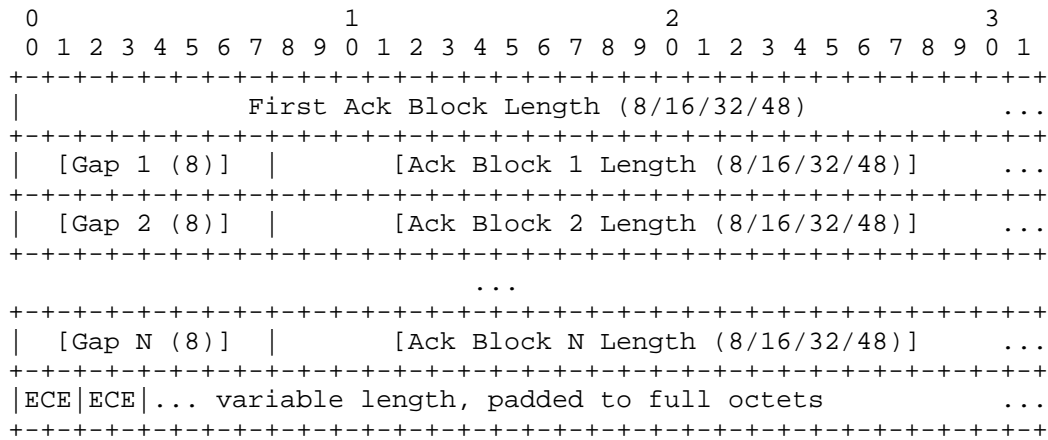


Figure 3: ECN field in ACK frame ACK block, alt 2

The third alternative encodes the number of bytes that are marked ECT(0), ECT(1) and CE with 32 bits each, the total extra overhead is thus 12 octets.

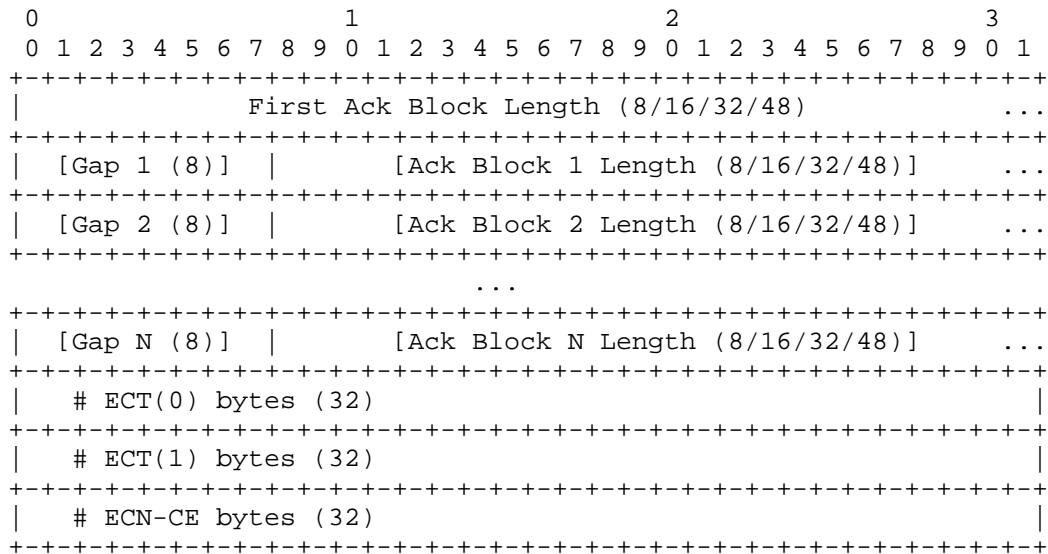


Figure 4: ECN field in ACK frame ACK block, alt 3

The fourth alternative use an extra byte to encode how many bits that encode each of the ECT/CE fields.

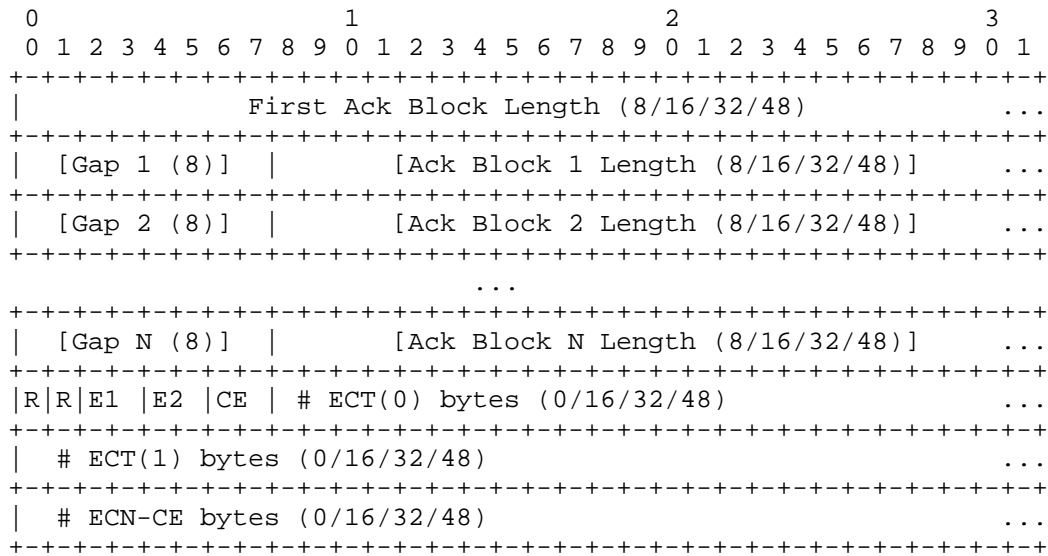


Figure 5: ECN field in ACK frame ACK block, alt 4

The E1, E2 and CE fields indicate the length of each encoding for the number of ECT(0), ECT(1) and ECN-CE marked bytes. This is encoded as:

- o 00: 0 bits
- o 01: 16 bits
- o 10: 32 bits
- o 11: 48bits

R indicates reserved bits.

There are pros and cons with the four alternatives:

- o Alt 1: Is very compact in the case where the ECN bits are largely unchanged. However in the worst case where received frames flip forth and back between ECT and CE then each frame will require at least 3 octets overhead (ECE, ACK block length, Gap).
- o Alt 2: Is quite compact as it only requires two bits encoding per frame. The additional overhead amounts to $\text{ceil}(N \cdot 2/8)$ octets where the N is the sum of the ACK block lengths. On the downside is that it is a less efficient format for the case that the ECN

bits are unchanged. One uncertainty is if STOP_WAITING frames could make this encoding bulky.

- o Alt 3: Has a fixed 12 octet overhead which may be beneficial as it gives a deterministic overhead. The possible drawback is that it is not possible to know exactly which frames have been remarked, something that can limit the ability to detect network ECN faults based on the method to transmit a pattern on ECT and CE marked packets.
- o Alt 4: Is a variation to Alt 3 but has a variable length encoding that should consume less space, especially in the cases that one of the ECT code points is not used and for the case that packets are only sporadically ECN-CE marked. This alternative also makes it unnecessary to use a bit in the ACK frame type to indicate the presence of an ECN field as this can be indicated in an efficient way with the one byte header in this format. E0=E1=CE = 00 indicates that the following ECT and CE fields are encoded with zero bits.

Which of the three formats above (or something else) that is the best alternative is subject to discussion.

2.4. Fallback in case of ECN fault

ECN can be subject to issues in network equipment, such as remarking to Not-ECN, remarking from ECT(0) to ECT(1) and vice versa or constant remarking to ECN-CE. Furthermore ECT marked packets may be discarded in the network. While these problems seem to be rare, see for instance [McQuistin-Perkins], it is still necessary to safeguard against such problems.

A peer should disable ECN for its outgoing packets if ECN fault is detected, it is however still possible for the other peer to use ECN.

TODO add more information as regards to how to detect network ECN faults. [ECN-fallback](expired) gives a few examples for fault detection. Examples on how to detect ECN faults include for instance the method to set ECT and CE for outgoing packets according to a given pattern.

Fallback in case of ECN faults is not an issue only for QUIC, it is here suggested that mechanisms for this is described in a non QUIC related draft, for instance in TSVWG.

2.5. OS socket specifics, access to the ECN bits

ECN support in QUIC comes with the additional challenge that it is necessary to somehow access the ECN bits in the IP headers. In TCP this is provided without major concerns as TCP is generally implemented in OS kernel space. QUIC can however be implemented both in user space or kernel space and is layered on top of UDP, which means that access to the ECN bits is not a given, instead various tricks are needed.

The text below is copy-pasted from [OHanlon].

"To set ECN on Linux, BSD and OSX one can use IP_TOS socket option, with the setsockopt() call, to set the relevant ECN bits of the TOS byte. On Windows one can use a similar technique though firstly one has to enable TOS byte setting by enabling a particular Registry key (DisableUserTOSSetting=0 (see <https://msdn.microsoft.com/en-us/library/windows/desktop/dd874008%28v=vs.85%29.aspx> One could also probably use the libpcap write functionality."

"To obtain the ECN bits from a packet one needs a mechanism to retrieve the ECN bits from each packet. On Linux, one needs to firstly set the IP_RECVTOS socket option on the receiving socket, and use the recvmsg() call to receive a packet, and then retrieve the TOS byte from the associated cmsg structure returned by the recvmsg() call. This still works with linux-4.2.3. On OSX/BSD there are no suitable socket options to retrieve the ECN/TOS bits and one cannot use raw sockets as they do not function for UDP/TCP sockets (they do work with ICMP), so one has to use alternatives such the bpf interface, or a REDIRECT socket. Whilst on Windows it seems that the only way to retrieve the ECN bits is via a raw socket, or custom NDIS driver, though it's possible there's an API I'm missing."

TODO: Write a more detailed description on how to implement ECN support in QUIC for different OS stacks.

2.6. Monitoring

A QUIC implementation should monitor the ECN functionality in order to provide input to e.g. service providers to improve ECN support in the networks. Items of interest are:

- o Black holes, ECT or CE marked packets are discarded.
- o Faulty remarking, e.g. ECT(0) is remarked to ECT(1) or Not-ECT.
- o Continuous CE marking, possible indication of faulty on/off ECN marking, but can also be an effect of severe congestion.

- o Degree of L4S support. L4S should generally give low queue latency. Estimation of one way queue delay for L4S enabled QUIC connections can be used to determine if there are congested nodes along the path that are not L4S capable.

3. IANA Considerations

T.B.D.

4. Open questions

A list of open questions:

- o Is it sufficient that one peer sends an ECN negotiation challenge frame?.
- o Should the ECN field in the ACK frame be mandatory ? (in which case it is not necessary to indicate its presence)
- o Should all packets be ECT or should there be special patterns to improve fault detection.
- o Write up a more detailed description on how to implement ECN support in QUIC for different OS stacks.
- o Determine which ECN echo encoding in the ACK frame is the best alternative.
- o Is a completely new ACK frame an alternative ?
- o How do STOP_WAITING frames affect the ECN echo overhead.
- o Outline possible connection migration actions
- o Are there any security implications with the smaller ECN negotiation frame ?

5. Security Considerations

T.B.D

6. Acknowledgements

The following persons have contributed with comments and suggestions for improvements: Mirja Kuehlewind, Koen De Schepper, Piers O'Hanlon, Michael Welzl, Marcelo Bagnulo Braun, Martin Duke

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

7.2. Informative References

- [Bagnulo] "Adding Explicit Congestion Notification (ECN) to TCP control packets and TCP retransmissions", <<https://tools.ietf.org/id/draft-bagnulo-tcpm-generalized-ecn-00.txt>>.
- [ECN-fallback] "A Mechanism for ECN Path Probing and Fallback", <<https://www.ietf.org/archive/id/draft-kuehlewind-tcpm-ecn-fallback-01.txt>>.
- [I-D.ietf-aqm-ecn-benefits] Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", draft-ietf-aqm-ecn-benefits-08 (work in progress), November 2015.
- [I-D.ietf-quic-transport] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.
- [I-D.ietf-tsvwg-ecn-experimentation] Black, D., "Explicit Congestion Notification (ECN) Experimentation", draft-ietf-tsvwg-ecn-experimentation-00 (work in progress), December 2016.
- [McQuistin-Perkins] "Is Explicit Congestion Notification usable with UDP?", Proceedings of the ACM Internet Measurement Conference, Tokyo, Japan, October 2015. DOI:10.1145/2815675.2815716, <<https://cisperkins.org/publications/2015/10/mcquistin2015ecn-udp.pdf>>.
- [OHanlon] "ECN support in different OS stacks", <<https://mailarchive.ietf.org/arch/msg/rmcat/rRKF3PVmFL2zHCplbOPKimqSsbM>>.

- [RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, DOI 10.17487/RFC6679, August 2012, <<http://www.rfc-editor.org/info/rfc6679>>.
- [RFC6789] Briscoe, B., Ed., Woundy, R., Ed., and A. Cooper, Ed., "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, DOI 10.17487/RFC6789, December 2012, <<http://www.rfc-editor.org/info/rfc6789>>.
- [RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <<http://www.rfc-editor.org/info/rfc7560>>.

Author's Address

Ingemar Johansson
Ericsson AB
Laboratoriegränd 11
Luleå 977 53
Sweden

Phone: +46 730783289
Email: ingemar.s.johansson@ericsson.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 1, 2017

I. Johansson
Ericsson AB
May 30, 2017

ECN support in QUIC
draft-johansson-quic-ecn-03

Abstract

This memo outlines the ECN (Explicit Congestion Notification) support in QUIC. The draft specifies the ECN negotiation and the ECN echo and in addition, different aspects of fallback in case of ECN failure as well as OS specific issues with ECN and monitoring for ECN capability. The intention is that most of the material ends up updating other new or existing QUIC protocol specifications, thus it may be possible that this draft does not warrant a working group status.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 1, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	2
2. Elements of ECN support	3
2.1. ECN negotiation	3
2.1.1. Challenge/Response	4
2.1.2. Determine degree of ECN support	5
2.2. ECN bits in the IP header, semantics	6
2.3. ECN echo	6
2.4. Fallback in case of ECN fault	7
2.5. OS socket specifics, access to the ECN bits	7
2.6. Monitoring	8
3. IANA Considerations	8
4. Open questions	8
5. Security Considerations	9
6. Acknowledgements	9
7. References	9
7.1. Normative References	9
7.2. Informative References	9
Author's Address	11

1. Introduction

ECN support in transport protocols is a fundamental feature that should be included in the QUIC specification as a mandatory element. ECN has the key benefit that it allows for non-destructive congestion

notification by network node, i.e packets are marked instead discarded. This is particularly beneficial for realtime applications with requirements on latency, ECN also has the benefit that it provides with a congestion signal that is unambiguous. The benefits with ECN is described in more detail in [I-D.ietf-aqm-ecn-benefits]. The ECN support should be implemented to support both present and future ECN, the latter is outlined in [I-D.ietf-tsvwg-ecn-experimentation], of particular interest is the ability to discriminate between classic ECN and L4S ECN by means of differentiation between the use of the ECT(0) and ECT(1) code points. This draft does however not delve into the details of the congestion control implementation.

2. Elements of ECN support

This draft covers the following aspects of ECN support:

- o ECN negotiation
- o ECN echo
- o ECN bits in the IP header, semantics
- o Fallback in case of ECN fault
- o OS socket specifics, access to the ECN bits
- o Monitoring

2.1. ECN negotiation

ECN support in QUIC needs to be negotiated. The reasons is that network elements may not support ECN and may either clear the ECN bits or simply discard packets that have the ECN bits set. In addition, a QUIC implementation may not have access to the ECN bits in the IP header due to OS dependent restrictions, investigations (Piers O'Hanlon) have indicated that this is in certain cases an asymmetric property, for instance while it is possible to set the ECN bits it is not possible to read them.

It is also required that the ECN negotiation does not interfere with the connection setup, in other words a failed ECN negotiation should not cause an extra roundtrip for the connection setup.

The suggested method in this draft is to send an ECN negotiation frame when connection setup is completed. Both peers MUST transmit the ECN negotiation frame. The ECN negotiation frame is shown below.

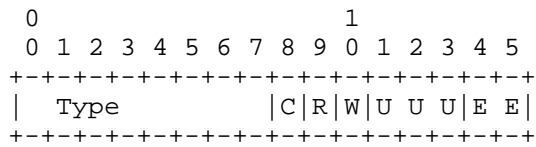


Figure 1: ECN negotiation frame

The 2nd byte contains the flags:

- o C: Challenge bit, indicates that the transmitted ECN negotiation frame is a challenge, if bit is not set then it is a response.
- o R: Possible to read ECN bits in IP header
- o W: Possible to write ECN bits in IP header
- o EE : Echo of ECN bits
- o U: Unused

The ECN negotiation has two steps.

- o Challenge/response
- o Determine degree of ECN support

2.1.1.1. Challenge/Response

A peer transmits the ECN negotiation frame with the R,W and EE bits in the 2nd byte set to '0' and the C bit set to '1'. This frame is echoed back with the flags set according to the degree of ECN support and with the ECN bits in the IP header of the received ECN negotiation frame copied to the EE field, the C bit is '0'. As both peers MUST transmit an ECN negotiation frame there will be a total of 4 ECN negotiation frames transmitted, two challenges and two responses.

An ECN negotiation frame should be transmitted in a unique packet, this to avoid that possible loss of ECN negotiation packets cause loss of other frames than the ECN negotiation frame.

The IP header for the ECN negotiation frame should set the ECN bits to CE '11'. When the corresponding response is received then an EE pattern of '11' indicates that ECN is likely supported in the network. This does not give a full guarantee that ECN is supported in the network. Monitoring of the ECN field in the ACK-frame serves to give further indication of ECN support once ECN is turned on.

An ECN negotiation is declared successful when an ECN negotiation response is received that indicates ECN support. A peer is not allowed to set ECT on outgoing data packets until a successful ECN negotiation is done. In other words it is only the ECN negotiation frame that is allowed to set the ECN bits in the IP header until ECN negotiation is concluded and successful.

A lack of an ECN negotiation response may indicate that the ECN challenge frame or the ECN response frame was lost or that a node in the network deliberately discards ECN-CE marked packets. The peer should transmit an additional ECN challenge within an RTO interval in case a negotiation response is not received, a maximum of retransmissions are attempted.

A failed challenge/response phase indicates that ECN should not be used in the connection. [NOTE, a special case is where one peer does not receive an ECN negotiation response but still receives ECT and CE marked packets from the other peer. It is T.B.D how this should be handled]

2.1.2. Determine degree of ECN support

If the ECN challenge/response is successful, the degree of ECN capability depends on how the R, W and EE bits are set.

- o R='1' and EE= '11': It is possible to set the ECN bits in outgoing packets.
- o R='0' or EE <> '11': ECN support is not certain as it is either not possible for remote peer to read the ECN bits or that the ECN bits are altered.
- o W='1' : It is meaningful to send ECN feedback
- o W='0' : It is not meaningful to send ECN feedback as the remote peer cannot set (write) the ECN bits in the IP header.

The mode mechanism in [RFC6679] can serve as input to a solution for the support of ECN in the case that OS ECN support is asymmetric. It is however unclear how a QUIC implementation can determine asymmetric ECN support in the underlying OS. For instance the method to send ECN marked packets to the local host to determine OS support does not reveal if the OS ECN support is asymmetric.

2.2. ECN bits in the IP header, semantics

The ECN bits in the IP header should be set according to the recommendations in [I-D.ietf-tsvwg-ecn-experimentation]. This means that the meaning of ECT(0) and ECT(1) differ.

2.3. ECN echo

The ECN echo should go into the ACK frame [I-D.ietf-quic-transport], this is beneficial as the ECN information can then use some of the already existing data in the ACK frame for improved efficiency.

The proposed alternative use one byte to encode how many bits that encode each of the ECT/CE fields.

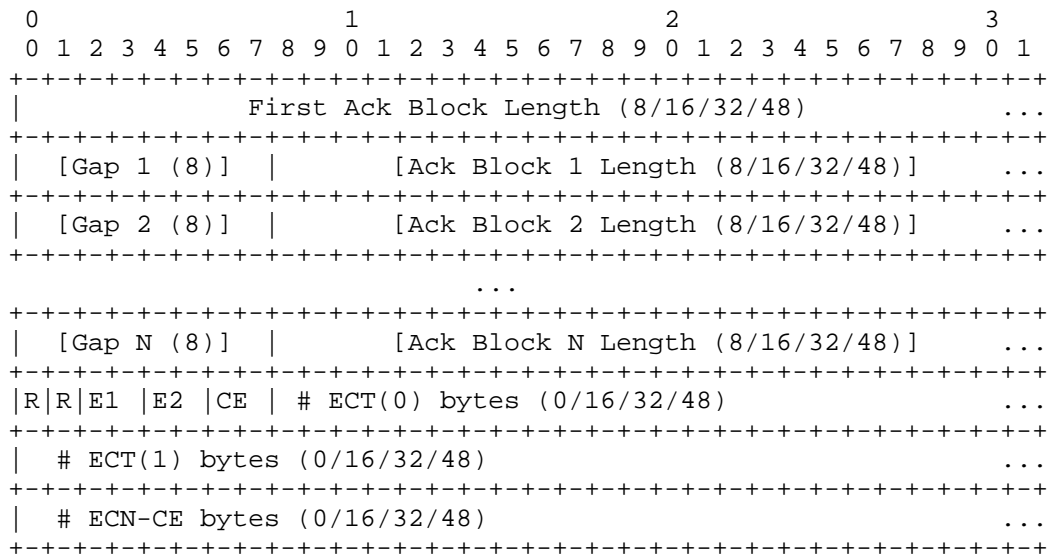


Figure 2: ECN field in ACK frame ACK block

The E1, E2 and CE fields indicate the length of each encoding for the number of ECT(0), ECT(1) and ECN-CE marked bytes. This is encoded as:

- o 00: 0 bits
- o 01: 16 bits
- o 10: 32 bits
- o 11: 48bits

R indicates reserved bits.

The proposed encoding enables flexible encoding of the ECN information, with a minimal 1 octet overhead for the cases where ECN is not supported by the connection.

2.4. Fallback in case of ECN fault

ECN can be subject to issues in network equipment, such as remarking to Not-ECN, remarking from ECT(0) to ECT(1) and vice versa or constant remarking to ECN-CE. Furthermore ECT marked packets may be discarded in the network. While these problems seem to be rare, see for instance [McQuistin-Perkins] and [APPLE-ECN], it is still necessary to safeguard against such problems.

A peer should disable ECN for its outgoing packets if ECN fault is detected, it is however still possible for the other peer to use ECN.

TODO add more information as regards to how to detect network ECN faults. [ECN-fallback](expired) gives a few examples for fault detection. Examples on how to detect ECN faults include for instance the method to set ECT and CE for outgoing packets according to a given pattern.

Fallback in case of ECN faults is not an issue only for QUIC, it is here suggested that mechanisms for this is described in a non QUIC related draft, for instance in TSVWG.

2.5. OS socket specifics, access to the ECN bits

ECN support in QUIC comes with the additional challenge that it is necessary to somehow access the ECN bits in the IP headers. In TCP this is provided without major concerns as TCP is generally implemented in OS kernel space. QUIC can however be implemented both in user space or kernel space and is layered on top of UDP, which means that access to the ECN bits is not a given, instead various tricks are needed.

The text below is copy-pasted from [OHanlon].

"To set ECN on Linux, BSD and OSX one can use IP_TOS socket option, with the setsockopt() call, to set the relevant ECN bits of the TOS byte. On Windows one can use a similar technique though firstly one has to enable TOS byte setting by enabling a particular Registry key (DisableUserTOSSetting=0 (see <https://msdn.microsoft.com/en-us/library/windows/desktop/dd874008%28v=vs.85%29.aspx> One could also probably use the libpcap write functionality."

"To obtain the ECN bits from a packet one needs a mechanism to retrieve the ECN bits from each packet. On Linux, one needs to firstly set the IP_RECVTOS socket option on the receiving socket, and use the `recvmsg()` call to receive a packet, and then retrieve the TOS byte from the associated `cmsg` structure returned by the `recvmsg()` call. This still works with linux-4.2.3. On OSX/BSD there are no suitable socket options to retrieve the ECN/TOS bits and one cannot use raw sockets as they do not function for UDP/TCP sockets (they do work with ICMP), so one has to use alternatives such the `bpf` interface, or a REDIRECT socket. Whilst on Windows it seems that the only way to retrieve the ECN bits is via a raw socket, or custom NDIS driver, though it's possible there's an API I'm missing."

TODO: Write a more detailed description on how to implement ECN support in QUIC for different OS stacks.

2.6. Monitoring

A QUIC implementation should monitor the ECN functionality in order to provide input to e.g. service providers to improve ECN support in the networks. Items of interest are:

- o Black holes, ECT or CE marked packets are discarded.
- o Faulty remarking, e.g. ECT(0) is remarked to ECT(1) or Not-ECT.
- o Continuous CE marking, possible indication of faulty on/off ECN marking, but can also be an effect of severe congestion.
- o Degree of L4S support. L4S should generally give low queue latency. Estimation of one way queue delay for L4S enabled QUIC connections can be used to determine if there are congested nodes along the path that are not L4S capable.

3. IANA Considerations

T.B.D.

4. Open questions

A list of open questions:

- o Is it sufficient that one peer sends an ECN negotiation challenge frame?.
- o Should all packets be ECT or should there be special patterns to improve fault detection.

- o Write up a more detailed description on how to implement ECN support in QUIC for different OS stacks.
- o Is a completely new ACK frame an alternative ?
- o Should amount on ECT(0), ECT(1) and CE marked bytes account for the IP+UDP headers or is it only the QUIC header + data that counts ?
- o Outline possible connection migration actions
- o Are there any security implications with the small ECN negotiation frame ?

5. Security Considerations

T.B.D

6. Acknowledgements

The following persons have contributed with comments and suggestions for improvements: Mirja Kuehlewind, Koen De Schepper, Piers O'Hanlon, Michael Welzl, Marcelo Bagnulo Braun, Martin Duke

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

7.2. Informative References

- [APPLE-ECN] Apple Inc., "TCP ECN: Experience with Enabling ECN on the Internet", <<https://www.ietf.org/proceedings/98/slides/slides-98-maprg-tcp-ecn-experience-with-enabling-ecn-on-the-internet-padma-bhooma-00.pdf>>.
- [Bagnulo] "Adding Explicit Congestion Notification (ECN) to TCP control packets and TCP retransmissions", <<https://tools.ietf.org/id/draft-bagnulo-tcpm-generalized-ecn-00.txt>>.

- [ECN-fallback]
"A Mechanism for ECN Path Probing and Fallback",
<<https://www.ietf.org/archive/id/draft-kuehlewind-tcpm-ecn-fallback-01.txt>>.
- [I-D.ietf-aqm-ecn-benefits]
Fairhurst, G. and M. Welzl, "The Benefits of using Explicit Congestion Notification (ECN)", draft-ietf-aqm-ecn-benefits-08 (work in progress), November 2015.
- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-03 (work in progress), May 2017.
- [I-D.ietf-tsvwg-ecn-experimentation]
Black, D., "Explicit Congestion Notification (ECN) Experimentation", draft-ietf-tsvwg-ecn-experimentation-02 (work in progress), April 2017.
- [McQuistin-Perkins]
"Is Explicit Congestion Notification usable with UDP?",
Proceedings of the ACM Internet Measurement Conference,
Tokyo, Japan, October 2015. DOI:10.1145/2815675.2815716",
<<https://csparks.org/publications/2015/10/mcquistin2015ecn-udp.pdf>>.
- [OHanlon] "ECN support in different OS stacks",
<<https://mailarchive.ietf.org/arch/msg/rmcat/rRKF3PVmFL2zHCplbOPKimqSsbM>>.
- [RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, DOI 10.17487/RFC6679, August 2012, <<http://www.rfc-editor.org/info/rfc6679>>.
- [RFC6789] Briscoe, B., Ed., Woundy, R., Ed., and A. Cooper, Ed., "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, DOI 10.17487/RFC6789, December 2012, <<http://www.rfc-editor.org/info/rfc6789>>.
- [RFC7560] Kuehlewind, M., Ed., Scheffenegger, R., and B. Briscoe, "Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback", RFC 7560, DOI 10.17487/RFC7560, August 2015, <<http://www.rfc-editor.org/info/rfc7560>>.

Author's Address

Ingemar Johansson
Ericsson AB
Laboratoriegården 11
Luleå 977 53
Sweden

Phone: +46 730783289
Email: ingemar.s.johansson@ericsson.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 9, 2017

M. Kuehlewind
B. Trammell
ETH Zurich
March 08, 2017

Applicability of the QUIC Transport Protocol
draft-kuehlewind-quic-applicability-00

Abstract

This document discusses the applicability of the QUIC transport protocol, focusing on caveats impacting application protocol development and deployment over QUIC. Its intended audience is designers of application protocol mappings to QUIC, and implementors of these application protocols.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 9, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Notational Conventions	2
2. The Necessity of Fallback	3
3. Zero RTT: Here There Be Dragons	3
4. Stream versus Flow Multiplexing	4
5. Prioritization	4
6. Graceful connection closure	5
7. Information exposure and the Connection ID	5
8. Use of Versions and Cryptographic Handshake	5
9. IANA Considerations	5
10. Security Considerations	5
11. Acknowledgments	6
12. References	6
12.1. Normative References	6
12.2. Informative References	6
Authors' Addresses	7

1. Introduction

QUIC [I-D.ietf-quic-transport] is a new transport protocol currently under development in the IETF quic working group, focusing on support of semantics as needed for HTTP/2 [I-D.ietf-quic-http] such as stream-multiplexing to avoid head-of-line blocking. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. This means the version of QUIC that is currently under development will integrate TLS 1.3 [I-D.ietf-quic-tls] to encrypt all payload data and most header information.

This document provides guidance for application developers that want to use the QUIC protocol without implementing it on their own. This includes general guidance for application use of HTTP/2 over QUIC as well as the use of other application layer protocols over QUIC. For specific guidance on how to integrate HTTP/2 with QUIC, see [I-D.ietf-quic-http].

In the following sections we discuss specific caveats to QUIC's applicability, and issues that application developers must consider when using QUIC as a transport for their application.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. The Necessity of Fallback

QUIC uses UDP as a substrate for userspace implementation and port numbers for NAT and middlebox traversal. While there is no evidence of widespread, systematic disadvantage of UDP traffic compared to TCP in the Internet [Edeline16], somewhere between three [Trammell16] and five [Swett16] percent of networks simply block UDP traffic. All applications running on top of QUIC must therefore either be prepared to accept connectivity failure on such networks, or be engineered to fall back to some other transport protocol. This fallback SHOULD provide TLS 1.3 or equivalent cryptographic protection, if available, in order to keep fallback from being exploited as a downgrade attack. In the case of HTTP, this fallback is TLS 1.3 over TCP.

These applications must operate, perhaps with impaired functionality, in the absence of features provided by QUIC not present in the fallback protocol. For fallback to TLS over TCP, the most obvious difference is that TCP does not provide stream multiplexing and therefore stream multiplexing would need to be implemented in the application layer if needed. Further, TCP by default does not support 0-RTT session resumption. TCP Fast Open could be used, but might not be supported by the far end or could be blocked on the network path. Note that there is some evidence of middleboxes blocking SYN data even if TFO was successfully negotiated (see [PaaschNanog]). Moreover, while encryption (in this case TLS) is inseparably integrated with QUIC, TLS negotiation over TCP can be blocked. In case it is RECOMMENDED to abort the connection, allowing the application to present a suitable prompt to the user that secure communication is unavailable.

We hope that the deployment of a proposed standard version of the QUIC protocol will provide an incentive for these networks to permit QUIC traffic. Indeed, the ability to treat QUIC traffic statefully as discussed in section 3.1 of [draft-kuehlewind-quic-manageability] would remove one network management incentive to block this traffic.

3. Zero RTT: Here There Be Dragons

QUIC provides for 0-RTT connection establishment (see section 3.2 of [I-D.ietf-quic-transport]). However, data in the frames contained in the first packet of a such a connection must be treated specially by the application layer. Since a retransmission of these frames resulting from a lost acknowledgment may cause the data to appear twice, either the application-layer protocol has to be designed such that all such data is treated as idempotent, or there must be some application-layer mechanism for recognizing spuriously retransmitted frames and dropping them.

Applications that cannot treat data that may appear in a 0-RTT connection establishment as idempotent MUST NOT use 0-RTT establishment. For this reason the QUIC transport SHOULD provide an interface for the application to indicate if 0-RTT support is in general desired or a way to indicate if data is idempotent.

4. Stream versus Flow Multiplexing

QUIC's stream multiplexing feature allows applications to run multiple streams over a single connection, without head-of-line blocking between streams, associated at a point in time with a single five-tuple. Streams are meaningful only to the application; since stream information is carried inside QUIC's encryption boundary, no information about the stream(s) whose frames are carried by a given packet is visible to the network.

Stream multiplexing is not intended to be used for differentiating streams in terms of network treatment. Application traffic requiring different network treatment SHOULD therefore be carried over different five-tuples (i.e. multiple QUIC connections). Given QUIC's ability to send application data on the first packet of a connection (if a previous connection to the same host has been successfully established to provide the respective credentials), the cost for establishing another connection are extremely low.

[EDITOR'S NOTE: For discussion: If establishing a new connection does not seem to be sufficient, the protocol's rebinding functionality (see section 3.7 of [I-D.ietf-quic-transport]) could be extended to allow multiple five-tuples to share a connection ID simultaneously, instead of sequentially.]

5. Prioritization

Stream prioritization is not exposed to the network, nor to the receiver. Prioritization can be realized by the sender and the QUIC transport should provide an interface for applications to prioritize streams [I-D.ietf-quic-transport].

Priority handling of retransmissions may be implemented in the transport layer and [I-D.ietf-quic-transport] does not specify a specific way how this must be handled. Currently QUIC only provides fully reliable stream transmission, and as such prioritization of retransmission is likely beneficial. For not fully reliable streams priority scheduling of retransmissions over data of higher-priority streams might not be desired. In this case QUIC could also provide an interface or derive the prioritization decision from the reliability level of the stream.

6. Graceful connection closure

[EDITOR'S NOTE: give some guidance here about the steps an application should take; however this is still work in progress]

7. Information exposure and the Connection ID

QUIC exposed some information to the network in the unencrypted part of the header. This is either because there is no encryption context established yet or because this information is intended to be consumed by the network. Some of these information can be optionally exposed (still under discussion). Given that exposing these information can have privacy implications, an application may indicate to not support exposure of certain information.

In case of the connection ID this can be the case if the application has additional information that the client is not behind a NAT and the server is not behind a load balancer, and therefore it is unlikely that the addresses will be re-bound.

8. Use of Versions and Cryptographic Handshake

Versioning in QUIC may change the whole protocol behavior, beside some header fields that have been declared to be fixed. As such a new or higher version of QUIC does not necessarily provide a better service but just a very different service, an application needs to be able to select which versions of QUIC it wants to use.

The use of a different encryption scheme than TLS1.3 or higher needs a new version of QUIC. [I-D.ietf-quic-transport] specifies requirements for the cryptographic handshake as currently realized by TLS1.3 and described in a separate specification [I-D.ietf-quic-tls]. This split is performed to enable light-weight versioning with different cryptographic handshakes.

9. IANA Considerations

This document has no actions for IANA.

10. Security Considerations

See the security considerations in [I-D.ietf-quic-transport] and [I-D.ietf-quic-tls]; the security considerations for the underlying transport protocol are relevant for applications using QUIC, as well.

Application developers should note that any fallback they use when QUIC cannot be used due to network blocking of UDP SHOULD guarantee the same security properties as QUIC; if this is not possible, the

connection SHOULD fail to allow the application to explicitly handle fallback to a less-secure alternative. See Section 2.

11. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

12. References

12.1. Normative References

- [I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.
- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

12.2. Informative References

- [draft-kuehlewind-quic-manageability]
Kuehlewind, M. and B. Trammell, "Manageability of the QUIC Transport Protocol", March 2017.
- [Edeline16]
Edeline, K., Kuehlewind, M., Trammell, B., Aben, E., and B. Donnet, "Using UDP for Internet Transport Evolution (arXiv preprint 1612.07816)", December 2016.
- [I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.

[PaaschNanog]

Paasch, C., "Network Ssupport for TCP Fast Open (NANOG 67 presentation)", June 2016.

[Swett16] Swett, I., "QUIC Deployment Experience at Google (IETF96 QUIC BoF presentation)", July 2016.

[Trammell16]

Trammell, B. and M. Kuehlewind, "Internet Path Transparency Measurements using RIPE Atlas (RIPE72 MAT presentation)", May 2016.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2017

M. Kuehlewind
B. Trammell
ETH Zurich
D. Druta
AT&T
March 09, 2017

Manageability of the QUIC Transport Protocol
draft-kuehlewind-quic-manageability-00

Abstract

This document discusses manageability of the QUIC transport protocol, focusing on caveats impacting network operations involving QUIC traffic. Its intended audience is network operators, as well as content providers that rely on the use of QUIC-aware middleboxes, e.g. for load balancing.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 10, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Notational Conventions	3
2. Features of the QUIC Wire Image	3
2.1. QUIC Packet Header Structure	3
2.2. Integrity Protection of the Wire Image	4
2.3. Connection ID and Rebinding	4
2.4. Packet Numbers	5
2.5. Greasing	5
3. Specific Network Management Tasks	5
3.1. Stateful Treatment of QUIC Traffic	5
3.2. Measurement of QUIC Traffic	6
3.3. DDoS Detection and Mitigation	6
3.4. QoS support and ECMP	7
3.5. Load balancing	8
4. IANA Considerations	8
5. Security Considerations	8
6. Acknowledgments	8
7. References	9
7.1. Normative References	9
7.2. Informative References	9
Authors' Addresses	10

1. Introduction

QUIC [I-D.ietf-quic-transport] is a new transport protocol currently under development in the IETF quic working group, focusing on support of semantics as needed for HTTP/2 [I-D.ietf-quic-http]. Based on current deployment practices, QUIC is encapsulated in UDP and encrypted by default. The current version of QUIC integrates TLS [I-D.ietf-quic-tls] to encrypt all payload data and most header information. Given QUIC is an end-to-end transport protocol, all information in the protocol header, even that which can be inspected, is is not meant to be mutable by the network, and will therefore be integrity-protected to the extent possible.

This document provides guidance for network operation on the management of QUIC traffic. This includes guidance on how to interpret and utilize information that is exposed by QUIC to the network as well as explaining requirement and assumptions that the QUIC protocol design takes toward the expected network treatment. It also discusses how common network management practices will be impacted by QUIC.

Of course, network management is not a one-size-fits-all endeavour: practices considered necessary or even mandatory within enterprise networks with certain compliance requirements, for example, would be impermissible on other networks without those requirements. This document therefore does not make any specific recommendations as to which practices should or should not be applied; for each practice, it describes what is and is not possible with the QUIC transport protocol as defined.

QUIC is at the moment very much a moving target. This document refers the state of the QUIC working group drafts as well as to changes under discussion, via issues and pull requests in GitHub current as of the time of writing.

1.1. Notational Conventions

The words "MUST", "MUST NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. Features of the QUIC Wire Image

In this section, we discuss those aspects of the QUIC transport protocol that have an impact on the design and operation of devices that forward QUIC packets. Here, we are concerned primarily with QUIC's unencrypted wire image, which we define as the information available in the packet header in each QUIC packet, and the dynamics of that information. Since QUIC is a versioned protocol, everything about the header format can change except the mechanism by which a receiver can determine whether and where a version number is present, and the meaning of the fields used in the version negotiation process. This document is focused on the protocol as presently defined in [I-D.ietf-quic-transport] and [I-D.ietf-quic-tls], and will change to track those documents.

2.1. QUIC Packet Header Structure

The QUIC packet header is under active development; see section 5 of [I-D.ietf-quic-transport] for the present header structure, and <https://github.com/quicwg/base-drafts/pull/361> for one current proposed redesign.

Currently the first bit of the QUIC header indicates the presence of a long header that exposed more information than the short. The long header is typically used during connection start or for other control processes while the short header will be used on mostly packets to limited unnecessary header overhead. The following information may be exposed in the packet header:

- o version number: The version number is present during version negotiation.
- o connection ID: The connection ID identifies the connection associated with a QUIC packet, for load-balancing and NAT rebinding purposes; see Section 2.3.
- o packet number: Every packet has an associated packet number; this packet number increases with each packet, and the least-significant bits of the packet number are present on each packet; see Section 2.4.
- o public reset indication: Public reset packets expose the fact that a connection is being torn down to devices along the path. The applicability of public reset is currently under discussion; see <https://github.com/quicwg/base-drafts/issues/353> and <https://github.com/quicwg/base-drafts/pull/20>.
- o key phase: To support 0-RTT session establishment, QUIC uses two key phases; the key phase of each packet must be exposed to support efficient reception.
- o additional flags: Additional flags for diagnostic use are also under consideration; see <https://github.com/quicwg/base-drafts/issues/279>.

[Editor's note: also further discuss which bits cannot change with versioning]

2.2. Integrity Protection of the Wire Image

As soon as the cryptographic context is established, all information in the QUIC header, including that exposed in the packet header, is integrity protected. Therefore, devices on path MUST NOT change QUIC packet headers, as alteration of header information would cause packet drop due to a failed integrity check at the receiver.

2.3. Connection ID and Rebinding

The connection ID in the QUIC packet header is used to allow routing of QUIC packets at load balancers on other than five-tuple information, ensuring that related flows are appropriately balanced together; and to allow rebinding of a connection after one of the endpoint's addresses changes - usually the client's, in the case of the HTTP binding. The connection ID is proposed by the server during connection establishment. A flow might change one of its IP addresses but keep the same connection ID, as noted in Section 2.1, and the connection ID may change during a connection as well; see

section 6.3 of [I-D.ietf-quic-transport]. See also <https://github.com/quicwg/base-drafts/issues/349> for ongoing discussion of the Connection ID.

2.4. Packet Numbers

The packet number field is always present in the QUIC packet header. The packet number exposes the least significant 32, 16, or 8 bits of an internal packet counter per flow direction that increments with each packet sent. This packet counter is initialized with a random 31-bit initial value at the start of a connection.

Unlike TCP sequence numbers, this packet number increases with every packet, including those containing only acknowledgment or other control information. Indeed, whether a packet contains user data or only control information is intentionally left unexposed to the network.

While loss detection in QUIC is based on packet numbers, congestion control by default provides richer information than vanilla TCP does. Especially, QUIC does not rely on duplicated ACKs, making it more tolerant of packet re-ordering.

2.5. Greasing

[Editor's note: say something about greasing if added to the transport draft]

3. Specific Network Management Tasks

In this section, we address specific network management and measurement techniques and how QUIC's design impacts them.

3.1. Stateful Treatment of QUIC Traffic

Stateful network devices such as firewalls use exposed header information to support state setup and tear-down. [I-D.trammell-plus-statefulness] provides a general model for in-network state management on these devices, independent of transport protocol. Features already present in QUIC may be used for state maintenance in this model. Here, there are two important goals: distinguishing valid QUIC connection establishment from other traffic, in order to establish state; and determining the end of a QUIC connection, in order to tear that state down.

1-RTT connection establishment, using a TLS handshake on stream 0, is detectable using heuristics similar to those used to detect TLS over TCP. 0-RTT connection establishment, however, provides no particular

heuristic for differentiation from random background traffic at this time.

Exposure of connection shutdown is currently under discussion; see <https://github.com/quicwg/base-drafts/issues/353> and <https://github.com/quicwg/base-drafts/pull/20>.

3.2. Measurement of QUIC Traffic

Passive measurement of TCP performance parameters is commonly deployed in access and enterprise networks to aid troubleshooting and performance monitoring without requiring the generation of active measurement traffic.

The presence of packet numbers on most QUIC packets allows the trivial one-sided estimation of packet loss and reordering between the sender and a given observation point. However, since retransmissions are not identifiable as such, loss between an observation point and the receiver cannot be reliably estimated.

The lack of any acknowledgement information or timestamping information in the QUIC packet header makes running passive estimation of latency via round trip time (RTT) impossible. RTT can only be measured at connection establishment time, and only when 1-RTT establishment is used.

Note that adding packet number echo (as in <https://github.com/quicwg/base-drafts/pull/367> or <https://github.com/quicwg/base-drafts/pull/368>) to the public header would allow passive RTT measurement at on-path observation points. For efficiency purposes, this packet number echo need not be carried on every packet, and could be made optional, allowing endpoints to make a measurability/efficiency tradeoff; see section 4 of [IPIM]. Note further that this facility would have significantly better measurability characteristics than sequence-acknowledgement-based RTT measurement currently available in TCP on typical asymmetric flows, as adequate samples will be available in both directions, and packet number echo would be decoupled from the underlying acknowledgment machinery; see e.g. [Ding2015]

Note in-network devices can inspect and correlate connection IDs for partial tracking of mobility events.

3.3. DDoS Detection and Mitigation

For enterprises and network operators one of the biggest management challenges is dealing with Distributed Denial of Service (DDoS) attacks. Some network operators offer Security as a Service (SaaS)

solutions that detect attacks by monitoring, analyzing and filtering traffic. These approaches generally utilize network flow data [RFC7011]. If any flows pose a threat, usually they are routed to a "scrubbing environment" where the traffic is filtered, allowing the remaining "good" traffic to continue to the customer environment.

This type of DDoS mitigation is fundamentally based on tracking state for flows (see Section 3.1) that have receiver confirmation and a proof of return-routability, and classifying flows as legitimate or DoS traffic. The QUIC packet header currently does not support an explicit mechanism to easily distinguish legitimate QUIC traffic from other UDP traffic. However, the first packet in a QUIC connection will usually be a client cleartext packet with a version field and a connection ID. This can be used to identify the first packet of the connection (also see <https://github.com/quicwg/base-drafts/issues/185>).

If the QUIC handshake was not observed by the defense system, the connection ID can be used as a confirmation signal as per [I-D.trammell-plus-statefulness]. In this case, similar as for all in-network functions that rely on the connection ID, a defense system can only rely on this signal for known QUIC's versions and if the connection ID is present (also see <https://github.com/quicwg/base-drafts/issues/293>).

Further, the use of a connection ID to support connection migration renders 5-tuple based filtering insufficient, and requires more state to be maintained by DDoS defense systems. However, it is questionable if connection migrations needs to be supported in a DDOS attack or if a defense system might simply rely on the fast resumption mechanism provided by QUIC. This problem is also related to these issues under discussion: <https://github.com/quicwg/base-drafts/issues/203> and <https://github.com/quicwg/base-drafts/issues/349>

3.4. QoS support and ECMP

QUIC does not provide any additional information on requirements on Quality of Service (QoS) provided from the network. QUIC assumes that all packets with the same 5-tuple {dest addr, source addr, protocol, dest port, source port} will receive similar network treatment. That means all stream that are multiplexed over the same QUIC connection require the same network treatment and are handled by the same congestion controller. If differential network treatment is desired, multiple QUIC connection to the same server might be used, given that establishing a new connection using 0-RTT support is cheap and fast.

QoS mechanisms in the network MAY also use the connection ID for service differentiation as usually a change of connection ID is bind to a change of address which anyway is likely to lead to a re-route on a different path with different network characteristics.

Given that QUIC is more tolerant of packet re-ordering than TCP (see Section 2.4), Equal-cost multi-path routing (ECMP) does not necessarily need to be flow based. However, 5-tuple (plus eventually connection ID if present) matching is still beneficial for QoS given all packets are handled by the same congestion controller.

3.5. Load balancing

[Editor's note: explain how this works as soon as we have decided who chooses the connection ID and when to set it. Related to <https://github.com/quicwg/base-drafts/issues/349>]

4. IANA Considerations

This document has no actions for IANA.

5. Security Considerations

Supporting manageability of QUIC traffic inherently involves tradeoffs with the confidentiality of QUIC's control information; this entire document is therefore security-relevant.

Some of the properties of the QUIC header used in network management are irrelevant to application-layer protocol operation and/or user privacy. For example, packet number exposure (and echo, as proposed in this document), as well as connection establishment exposure for 1-RTT establishment, make no additional information about user traffic available to devices on path.

At the other extreme, supporting current traffic classification methods that operate through the deep packet inspection (DPI) of application-layer headers are directly antithetical to QUIC's goal to provide confidentiality to its application-layer protocol(s); in these cases, alternatives must be found.

6. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

7. References

7.1. Normative References

- [I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.
- [I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.

7.2. Informative References

- [Ding2015]
Ding, H. and M. Rabinovich, "TCP Stretch Acknowledgments and Timestamps - Findings and Implications for Passive RTT Measurement (ACM Computer Communication Review)", July 2015.
- [draft-kuehlewind-quic-applicability]
Kuehlewind, M. and B. Trammell, "Applicability of the QUIC Transport Protocol", March 2017.
- [I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-01 (work in progress), January 2017.
- [I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-02 (work in progress), December 2016.
- [IPIM]
Allman, M., Beverly, R., and B. Trammell, "In-Protocol Internet Measurement (arXiv preprint 1612.02902)", December 2016.

[RFC7011] Claise, B., Ed., Trammell, B., Ed., and P. Aitken,
"Specification of the IP Flow Information Export (IPFIX)
Protocol for the Exchange of Flow Information", STD 77,
RFC 7011, DOI 10.17487/RFC7011, September 2013,
<<http://www.rfc-editor.org/info/rfc7011>>.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Dan Druta
AT&T

Email: dd5826@att.com